Università degli Studi di Pisa

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di laurea:

Uno strumento per la simulazione di reti di sensori multi-applicazione basato su Tossim

Relatore: Prof.ssa Cinzia Bernardeschi

Relatore: Prof. Giuseppe Anastasi

Candidato: Mattia Rausa

Anno Accademico 2003/2004

ai miei genitori . . .

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

Le reti di sensori rappresentano un nuovo modello di rete che sfrutta le capacità di elaborazione dei nodi, le dimensioni ridotte dei dispositivi e la possibilità di comunicare via radio per poter essere inserite in un qualsiasi contesto dove vi è la necessita di monitorare un sistema. Queste reti condividono alcuni aspetti delle reti *ad hoc* attualmente esistenti ma possiedono caratteristiche tali da imporre una revisione di molti degli algoritmi e protocolli fino ad ora utilizzati.

Tossim è un simulatore per reti di sensori che permette di testare le applicazioni su un numero considerevole di nodi e fornisce la possibilità di controllare l'ambiente di simulazione in modo semplice.

Lo scopo di questa tesi è quello di estendere le funzionalità di Tossim permettendo la simulazione di applicazioni diverse sui nodi sensore che cooperano tra loro. Lo strumento sviluppato utilizza un modulo che coordina varie istanze di Tossim per mettere in comunicazione nodi appartenenti ad applicazioni diverse, inoltre lascia inalterate le funzionalità di Tossim a disposizione dell'utente esterno. Per favorire la portabilità, lo strumento è stato implementato in Java e sono state ridotte al minimo le modifiche a Tossim.

Introduzione

Negli ultimi anni, i passi in avanti nella miniaturizzazione, nella progettazione di circuiti a basso consumo e l'ottimo livello di efficienza raggiunto dai dispositivi di comunicazione a onde radio, hanno reso possibile, una nuova prospettiva tecnologica: le reti di sensori. Queste reti combinano le capacità di raccogliere informazioni dall'esterno, effettuare delle elaborazioni e comunicare attraverso un ricetrasmettitore, per realizzare una nuova forma di rete, che può essere installata all'interno di un ambiente fisico, sfruttando le dimensioni ridotte dei dispositivi che la compongono e il loro basso costo.

Questi dispositivi vengono collocati all'interno dell'area dove si verifica il fenomeno che si intende analizzare o nelle sue immediate vicinanze. I nodi sensore per svolgere il loro compito sono dotati di un microprocessore, di un modulo per le comunicazioni radio e di una serie di dispositivi elettronici (i sensori) in grado di percepire le più piccole modificazioni dell'ambiente esterno, quali ad esempio temperatura, luminosità, accelerazione, etc. Tipicamente un nodo sensore svolge le seguenti operazioni: raccoglie i dati provenienti dall'osservazione locale del fenomeno, li elabora, aggrega i risultati con le informazioni provenienti da punti diversi dell'area e infine trasmette i dati verso una o più stazioni base inviandoli ai nodi vicini secondo un protocollo *multi-hop*.

Sulla base di tale tecnologia nuovi tipi di applicazioni diventano possibili. Le reti di sensori possono essere impiegate nel controllo ambientale [7], come l'individuazione degli incendi boschivi, ma anche essere installate su ponti e costruzioni per studiare il fenomeno dei terremoti [8]; possono essere utilizzate per compiti di sorveglianza come il riconoscimento di intrusioni in aree protette; possono essere inseriti nei macchinari dove non sono possibili collegamenti via cavo tra i sensori [10], oppure attaccati al corpo umano per il monitoraggio a distanza dei dati fisiologici di un paziente [11]. Questi sono solo alcuni esempi, che stanno a dimostrare come questa tecnologia può essere usata con successo. Obbiettivi futuri sono la riduzione delle dimensioni dei sensori fino a pochi millimetri cubici e la minimizzazione del loro costo,

che non dovrebbe superare il dollaro.

Una serie di servizi per supportare vari tipi di applicazione sono stati sviluppati per le reti di sensori. Tra i più importanti citiamo i meccanismi legati alla sincronizzazione temporale [12], che forniscono alla rete nella sua interezza una nozione consistente del tempo; la localizzazione [13], che permette ai nodi di conoscere la loro posizione all'interno di un area; l'instradamento affidabile dei pacchetti, che si rende necessario quando la tipologia della rete cambia dinamicamente [14] [15]; il servizio di tracciamento degli oggetti in movimento all'interno della rete, etc. La progettazione e l'implementazione di applicazioni basate su una piattaforma così versatile ha sollevato inoltre una serie di questioni in diversi campi della ricerca. Per esempio, quando una rete di sensori viene utilizzata in applicazioni per il controllo di sistemi, tale rete dovrà, per definizione, interagire con il sistema. Si presenta in questo caso, il problema del controllo distribuito applicato alle reti di sensori [17], che rientra in quello più generale degli algoritmi distribuiti. Il comune denominatore di ogni studio comunque, è rappresentato dalla longevità di queste reti. In molti casi i nodi sensore potranno essere alimentati unicamente da batterie esauribili, si pone quindi il problema di ricercare soluzioni che preservino al massimo la quantità di energia disponibile [18].

Molte delle soluzioni proposte sono state testate e validate attraverso l'analisi di simulazioni effettuate al computer, per le quali vengono adoperati strumenti specifici per rappresentare in modo fedele le diverse realtà [1] [2] [5] [24]. Per verificare il corretto funzionamento di un'applicazione per reti di sensori basati sul sistema operativo TinyOS è disponibile il simulatore Tossim [3] [4]. Tossim consente di simulare reti con la stessa applicazione su tutti i nodi sensore. Lo scopo di questa tesi è di estendere le funzionalità di Tossim per simulare applicazioni diverse su nodi che cooperano tra loro. Benché questo strumento sia *open-source* abbiamo preferito ridurre al massimo le modifiche al codice che lo implementa privilegiando il concetto di modularità e portabilità del software. Ciò è reso possibile dalle capacità offerte dal simulatore che permette ad un'applicazione esterna di interagire

con lo stesso tramite un protocollo ben definito. In effetti Tossim è corredato da un applicazione esterna chiamata Tinyviz che visualizza graficamente lo stato della simulazione e fornisce all'utente dei comandi per interagire e modificare alcuni parametri della simulazione. Il nostro lavoro si è focalizzato sulla realizzazione di un modulo che si interpone tra le varie istanze del Tossim e il Tinyviz controllandone il comportamento in modo da rendere realistico l'ambiente simulato. Il modulo proposto è stato implementato utilizzando il linguaggio Java in modo da favorirne la portabilità ed infine è stato realizzato un *plug-in* per il visualizzatore grafico per poter distinguere i nodi appartenenti ad applicazioni diverse.

Il lavoro di tesi è così strutturato:

- Capitolo 1. Descrive gli aspetti più interessanti di una reti di sensori.
- Capitolo 2. Analizza le caratteristiche essenziali di una tipica piattaforma hardware e ne presenta alcune realizzazioni.
- Capitolo 3. Descrive in dettaglio il sistema operativo TinyOS e alcuni aspetti del linguaggio di programmazione NesC.
- Capitolo 4. Descrive le caratteristiche dell'ambiente di simulazione Tossim.
- Capitolo 5. Illustra l'architettura software del nostro progetto.
- Capitolo 6. Analizza in dettaglio le scelte implementative e riporta alcune considerazioni sulla validità dello strumento sviluppato.

Capitolo 1

Le reti di sensori

Le reti di sensori rappresentano l'ultima frontiera della ricerca nel campo dei sistemi cosiddetti *embedded*. Questi dispositivi sono stati concepiti per interagire con l'ambiente che li circonda, ma mentre fino ad ora la possibilità di cooperare con altri sistemi era limitata in molti casi dalle connessioni via cavo, al giorno d'oggi i progressi nelle comunicazioni senza fili hanno eliminato questo vincolo facilitandone la collocazione all'interno dell'ambiente da monitorare. Si vengono a formare in questo modo delle vere e proprie reti nelle quali ciascun nodo assolve uno specifico compito e collabora con gli altri nodi della rete per raggiungere determinati obbiettivi.

1.1 Reti di sensori e reti *ad hoc*

Una rete di nodi sensori condivide molti elementi con le reti *ad hoc* esistenti. Ma esistono numerose differenze e specificità che hanno suscitato una particolare attenzione da parte del mondo della ricerca. Alcuni dei punti che rendono le reti di sensori diverse dalle altre sono:

Applicazioni

L'enorme varietà delle applicazioni possibili richiede soluzioni diversificate. Ad esempio, rispetto ai protocolli da utilizzare, la tolleranza ai guasti deve essere tenuta in considerazione se i sensori operano in

un ambiente critico. Ma esistono delle applicazioni in cui i guasti si verificano raramente e quindi quest'aspetto non rappresenta un vincolo.

Scalabilità

Potenzialmente, il numero di unità che formano una rete varia da qualche decina alle migliaia di nodi (più delle reti attuali) e richiedono soluzioni fortemente scalabili.

Energia

Spesso i nodi sono alimentati unicamente da batterie ed è quindi necessario adottare misure che prolunghino al massimo la vita dei sensori.

Autoconfigurazione

Il comportamento di un nodo, in alcuni casi, deve variare a seconda dello stato in cui si trova, come energia, posizione, ruolo, etc. In pratica i nodi devono autoconfigurarsi e rendere noti i parametri scelti ai vicini.

Centralità dei dati

Rispetto alle comuni reti l'importanza di un nodo è ridotta notevolmente, quello che veramente interessa è l'osservazione del fenomeno. Se un sensore si guasta è molto probabile che ciò non influisca sulle prestazioni dell'intera rete, in quanto la ridondanza è una caratteristica delle reti di sensori.

Semplicità

Per poter realizzare nodi sensore a basso costo, basso consumo e dimensioni ridotte, sia l'hardware che il software devono essere il più semplici possibili.

Fattore ambientale

L'ambiente in cui una rete di sensori si troverà a lavorare impone delle scelte sostanziali, in relazione al tipo di fenomeno da osservare. Ad esempio il tipo di traffico prodotto, come banda minima, frequenza dei picchi di traffico, vincoli real-time, etc., influisce sulla scelta tra un protocollo ed un altro.

1.2 Una rete di sensori

Una rete di sensori è un insieme, eventualmente eterogeneo, di nodi sensore che forma una predeterminata topologia. I nodi sensore possono essere equipaggiati con dispositivi in grado di rilevare una grande varietà di condizioni ambientali, come temperatura, umidità, movimento dei veicoli, luminosità, pressione, livello di rumore, presenza o assenza di certi tipi di oggetti, grado di stress meccanico, valori istantanei di velocità, direzione e dimensioni di un oggetto.

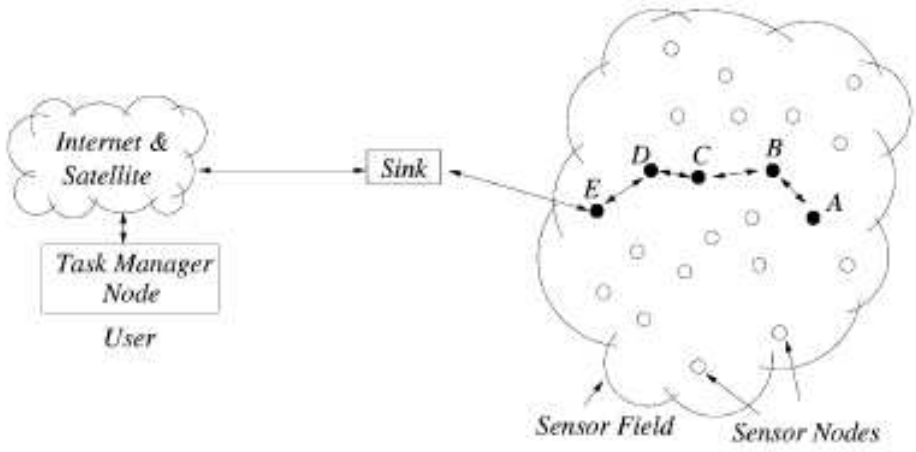

Figura 1.1: Nodi sensore disposti in un area.

I nodi sono collocati all'interno dell'area in cui si verifica il fenomeno che si vuole analizzare, oppure, dove non è possibile, nelle sue immediate vicinanze. Questa attività può essere ripetuta quando c'è, per esempio, la necessità di rimpiazzare delle unità che hanno esaurito l'energia della loro batteria o sono state danneggiate. Generalmente ogni nodo viene associato ad un oggetto, a un essere vivente o a un luogo, vengono posti in pratica nei punti chiave del fenomeno. Il numero di sensori che compongono la rete varia da qualche decina a svariate migliaia. Le informazioni raccolte vengono poi inviate verso una stazione base, passando da un nodo ad un altro secondo un

protocollo *multi-hop*, ed infine trasferite, via internet o via satellite, verso un centro di raccolta. Si possono prevedere configurazioni con più stazioni base, eventualmente mobili (ad esempio un utente con un PDA oppure un aereo in volo), e in questo caso, i nodi coinvogliano i dati verso un sottoinsieme delle stazioni stesse.

In altri casi le reti sono completamente isolate. Ciò può accadere quando i nodi sono equipaggiati con attuatori e controllori, e programmati per svolgere determinate attività.

1.3 Applicazioni

La grande versatilità delle reti di sensori favorisce il loro impiego in diversi ambiti:

Ambiente

Alcune applicazioni riguardano l'osservazione degli spostamenti di animali, uccelli, specie marine, insetti; il monitoraggio delle condizioni ambientali a cui sono sottoposte le colture e gli allevamenti di bestiame; il riconoscimento degli incendi boschivi; la ricerca metereologica e geofisica; il riconoscimento delle inondazioni; la mappatura della biocomplessità di un ambiente; lo studio dell'inquinamento. Alcune implementazioni di questi esempi li troviamo nel sistema per il riconoscimento delle inondazioni denominato ALERT [25], utilizzato negli Stati Uniti; nella mappatura della biocomplessitá, con il sistema installato nella James Reserve in Southern California [26]; nell'osservazione degli spostamenti degli uccelli migratori, con il progetto denominato Great Duck Island Habitat Monitoring [27]

Applicazioni militari

Le reti di sensori possono essere parte integrante dei sistemi militari di comando, controllo, comunicazione, elaborazione, inteligence, sorveglianza, riconoscimento ed individuazione. La rapidità di collocazione,

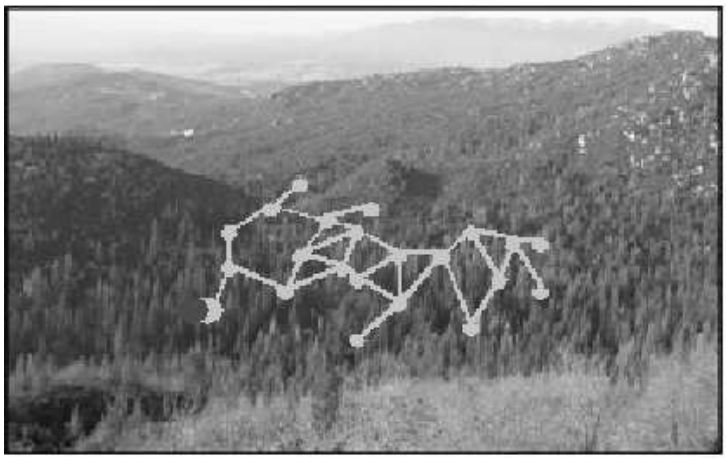

Figura 1.2: Nodi sensore disposti in un bosco.

l'auto-organizzazione e la tolleranza ai guasti sono caratteristiche che rendono queste reti particolarmente adatte all'impiego in campo militare. Dal momento che i nodi possono popolare densamente una qualsiasi area ed hanno un costo ridotto, la distruzione di alcuni nodi in seguito ad un azione ostile non ne inibisce il funzionamento globale. Alcune applicazioni sono il monitoraggio delle forze alleate; il riconoscimento del nemico; l'individuazione degli obbiettivi; la sorveglianza di intere aree; il riconoscimento e l'individuazione di attacchi nucleari, biologici e chimici.

Sanità

Alcune applicazioni nella sanità sono: la realizzazione di interfacce per i disabili; il monitoraggio integrato dei pazienti; la somministrazione di farmaci negli ospedali; il telemonitoraggio dei dati fisiologici di un individuo; il tracciamento e il monitoraggio dei pazienti e del personale medico negli ospedali. Un esempio di telemonitoraggio dei dati fisiologici di un paziente è dato dal sistema Helat Smart Home, realizzato dalla Facoltà di medicina di Grenoble - Francia [28].

Uso domestico

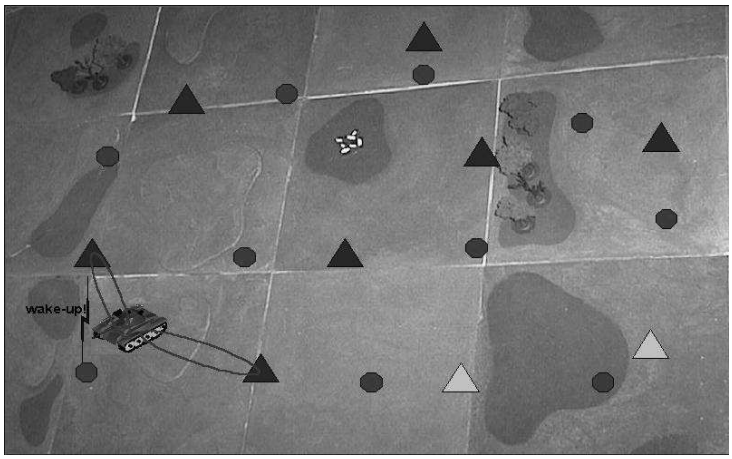

Figura 1.3: Sorveglianza di un campo di battaglia.

I nodi sensore possono essere inseriti negli apparecchi elettrici, come forni a microonde, frigoriferi, VCR [29]. Questi possono interagire tra loro e con una rete esterna, via internet o via satellite e permettono all'utente di comandare facilmente gli elettrodomestici a distanza. Più in generale, le reti di sensori possono essere utilizzate per realizzare ambienti intelligenti non solo in casa ma anche negli uffici. Un esempio è il Residential Laboratory della Georgia Institute of Tecnology [30]. Gli obbiettivi di questo laboratorio sono l'affidabilità, la persistenza e la trasparenza dell'automazione.

Applicazioni commerciali

Altre possibili applicazioni sono: la realizzazione di tastiere virtuali; l'analisi del comportamento dei materiali sottoposti a stress meccanico, la qualità di un prodotto; guida e controllo dei robot nelle industrie automatizzate.

1.4 Parametri che influenzano la progettazione di una rete di sensori

Per una rete di sensori esistono una serie di fattori definiti critici, perché costituiscono le linee guida da seguire nella progettazione di algoritmi e protocolli e possono essere utilizzati come termine di paragone per differenti soluzioni. Tra i più importanti citiamo:

Tolleranza ai guasti

Alcuni nodi possono guastarsi o rimanere bloccati a causa di danni fisici, interferenze ambientali oppure mancanza di energia. Tuttavia il malfunzionamento di una unità non dovrebbe avere effetti collaterali sulle funzionalità dell'intera rete. Comunque, la probabilità che un nodo si guasti dipende in grossa parte dal tipo di applicazione per cui vengono impiegati. I sensori utilizzati in applicazioni militari, ad esempio, sono soggetti a malfunzionamenti più di quelli utilizzati in ambito domestico [31].

Scalabilità

Il numero di sensori che possono comporre una rete variano da qualche decina alle centinaia di migliaia. I protocolli e gli algoritmi devono essere in grado di supportare un numero così elevato di nodi.

Costi di produzione

Data l'enorme quantità di sensori che alcune applicazioni richiedono, il costo per unità non dovrebbe pesare sul costo complessivo di una rete di sensori. La speranza è quella di non superare il dollaro per un nodo, ma attualmente il prezzo del sistema radio più economico, il Bluetooth, si aggira sui dieci dollari. In più bisogna considerare che un nodo può essere equipaggiato con altri dispositivi, quali GPS, alimentatori, attuatori, che ne fanno lievitare il costo [32].

Dimensioni

Un nodo sensore è generalmente costituito da un processore, un ricetra-

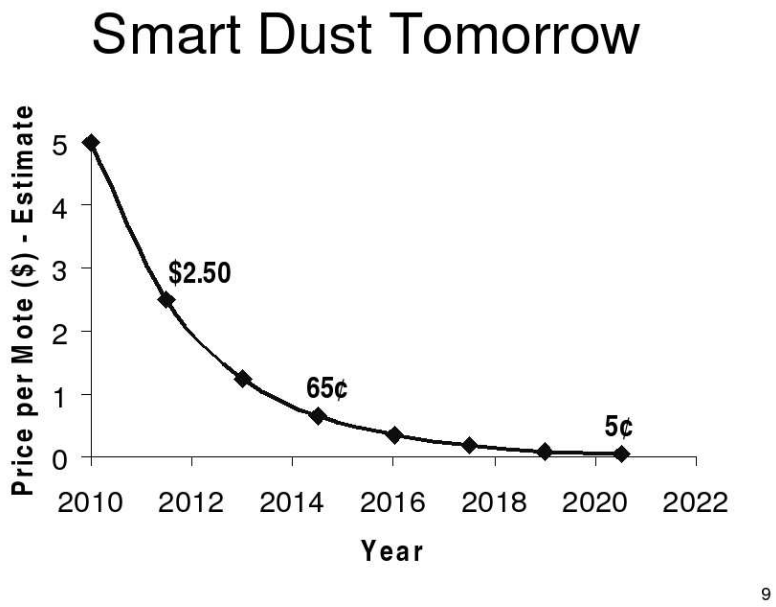

Figura 1.4: Previsione dei costi.

smettitore, una batteria, un convertitore analogico/digitale, un modulo di memoria, e dei dispositivi per l'acquisizione di dati ambientali. Tutti questi moduli devono essere inseriti in una scatola che in alcuni casi non dovrebbe superare le dimensioni di un centimetro cubo se, per esempio, deve rimanere sospesa in aria.

Figura 1.5: Dimensioni di un nodo sensore.

Consumo energetico

Un nodo sensore essendo un dispositivo microelettronico deve essere alimentato con una limitata sorgente di energia (< 0.5 A, 1.2 V). In alcuni casi i nodi possono essere equipaggiati con delle celle solari, ma quando ciò non è possibile bisogna adottare delle misure per ridurre al massimo i consumi. Principalmente si tratta di tenere spenta la radio il più a lungo possibile, perché tra tutti, è il dispositivo che consuma la maggior quantità di energia.

1.5 Progettazione di una rete di sensori

In questo paragrafo descriveremo le problematiche che un progettista deve affrontare nella realizzazione di una rete di sensori, per dare una visione più completa delle caratteristiche di questa nuova tecnologia. Data la vastità dell'argomento e gli scopi che si prefissa questa tesi, esamineremo solo gli aspetti principali rimandando ad altri testi l'approfondimento dell'argomento.

1.5.1 Protocolli di comunicazione

I protocolli di comunicazione rappresentano una delle aree di ricerca più attive nell'ambito delle reti di sensori. I maggiori sforzi si sono concentrati nella realizzazione di protocolli MAC efficienti dal punto di vista energetico e di protocolli di routing che offrano soluzioni efficaci per tutte le esigenze. Dei livelli di cui è composto il modello ISO/OSI, quelli coinvolti nella progettazione di protocolli di rete per una rete di sensori sono il livello fisico, il livello data-link ed il livello di rete, di cui ne daremo una breve descrizione. Il livello di trasporto invece viene preso poco in considerazione dalla comunità di ricerca e quindi non verrà discusso.

Livello Fisico

Il livello fisico si occupa di impostare la frequenza, di generare la portante e del riconoscimento del segnale e infine della modulazione e codifica dei dati. Ma rispetto alle classiche trasmissioni radio, in questo caso il principale problema è come trasmettere i dati spendendo la quantità minima di energia, tenendo conto di tutti i costi collegati (overhead, possibili ritrasmissioni, etc). Molti lavori si concentrano sulla modulazione efficiente del segnale dal punto di vista energetico [33] [34].

Livello data-link

Il livello data-link è responsabile della multiplazione dei flussi di dati, del riconoscimento dei frame, del controllo dell'accesso al mezzo e del controllo dell'errore. Di questi aspetti forse il più interessante è la progettazione del protocollo MAC.

I protocolli MAC in una rete di sensori devono svolgere due compiti:

- dal momento che i nodi sensore vengono sparsi in un area, il protocollo deve stabilire quali sono i nodi vicini con cui è in grado di comunicare (si tenga conto che in alcune applicazioni i nodi sono in grado di muoversi all'interno di un area);
- condividere il mezzo trasmissivo nel modo più efficiente ed equo possibile, dato che la ricetrasmittente è l'unità che consuma più energia. In pratica il protocollo deve cercare di tenerla spenta il più a lungo possibile quando non è in grado di inviare dati.

Livello di rete

Il livello di rete condivide alcune caratteristiche con le attuali reti ad hoc, ma le reti di sensori richiedono nuove soluzioni per quanto riguarda la scalabilità il consumo energetico, la sicurezza, il real-time, e la mobilità. Per quanto riguarda la sicurezza, ad esempio, bisogna tener conto della:

- vulnerabilità del canale (iniezione di messaggi falsi),
- vulnerabilità dei nodi (cancellazione o modifica dei messaggi, reinstradamento),
- assenza di infrastrutture (es. non esiste un'autorità centrale di certificazione),
- dinamicità della topologia della rete (difficile riconoscere i movimenti dei nodi dagli attacchi).

In effetti un attaccante potrebbe avere capacità simili (hardware, ..), potrebbe sostituire i nodi oppure potrebbe disporre di sistemi di comunicazione a lungo raggio.

1.5.2 Servizi

Un'applicazione per reti di sensori richiede una serie di servizi tali da fornire informazioni, o attuare comportamenti, necessari allo svolgimento dei compiti per le quali sono state progettate. In questo paragrafo analizzeremo il servizio di localizzazione, di gestione dei consumi energetici e di sincronizzazione temporale. Comunque esistono altri servizi che un'applicazione può richiedere ma che accenneremo solamente. Ad esempio, le applicazioni destinate alla sorveglianza di un ambiente necessitano di meccanismi che garantiscano un certo grado di copertura dell'area da monitorare Sempre nello stesso ambito un applicazione deve essere in grado di effettuare il tracciamento degli oggetti in movimento all'interno della rete.

Localizzazione

La localizzazione è il processo con il quale un nodo determina la sua posizione all'interno di un area. I fattori da tener conto nella progettazione di tali sistemi sono:

- costo (dell'infrastruttura, ad es. GPS),

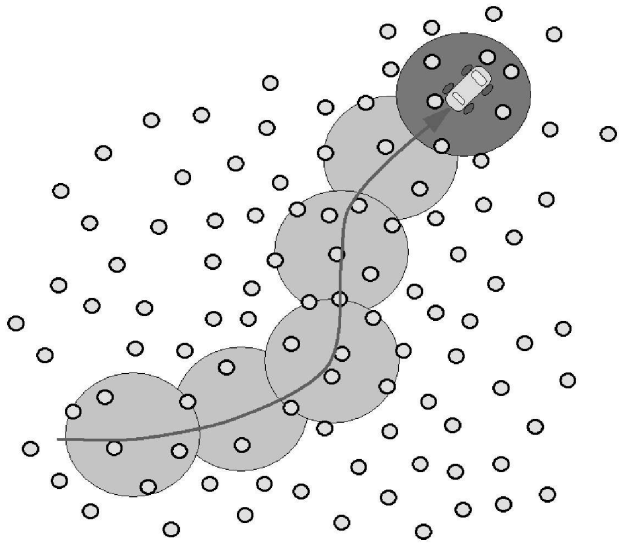

Figura 1.6: Servizio di tracciamento di veicoli.

- accuratezza (in metri o rispetto al raggio della copertura radio),
- velocità con cui avviene il processo (secondi, minuti),
- overhead (per quanto riguarda il consumo di energia e i messaggi inviati).

Gestione dei consumi energetici

Nel caso in cui i nodi non hanno una disponibilità illimitata di energia bisogna adottare soluzioni che ne incrementino il tempo di vita dell'intero sistema. Il problema si riassume nel saper quando spegnere e riaccendere dispositivi come:

- microcontrollore,
- memoria,
- radio,
- altro.

La gestione dei consumi riguarda sia l'hardware sia il software. È necessario impiegare dispositivi elettronici a basso consumo e protocolli e algoritmi efficienti dal punto di vista energetico. Tipicamente si cerca di tenere accesi i nodi minimi necessari a garantire che ogni nodo attivo possa comunicare con la stazione base e in grado di coprire l'area che si intende analizzare. Nel caso del nodo sensore denominato EYES, equipaggiato da un microprocessore Texas Instruments si hanno i seguenti valori:

- CPU modalità *active* 1.2 mA
- CPU modalità *sleep* 1.6 μA
- Radio in trasmissione 10 mA
- Radio in ricezione 4 mA
- Radio sleep 20 μA

Sincronizzazione temporale

La sincronizzazione temporale fornisce a tutti i nodi della rete una nozione consistente del tempo con una certa precisione. Ad esempio, può risultare necessario che le osservazioni vengano annotate con il corretto tempo, che i nodi siano sincronizzati per i cicli di riposo, etc. Alcuni problemi intrinseci delle reti di sensori come le limitare risorse di energia, banda, elaborazione, memoria, combinate con l'alta densità dei nodi rendono i tradizionali algoritmi inutilizzabili per queste reti. I fattori che influenzano la progettazione di questi protocolli sono:

- precisione (secondo, millisecondo, etc),
- costo energetico,
- durata (frequenza con cui viene attivato il processo di sincronizzazione),
- scopo (due nodi vicini o l'intera rete),
- costo.

1.5.3 Supporto alle applicazioni ad alto livello

Nel precedente paragrafo sono stati illustrati i servizi che si rendono necessari per garantire il funzionamento delle applicazioni. In questo paragrafo descriviamo invece un ulteriore livello che compare nella stratificazione del modello ISO/OSI per le reti di sensori. Tale livello, chiamato supporto ad alto livello per le applicazioni, precede lo strato delle applicazioni.

Database

Un approccio interessante , vede le reti di sensori come un database che può essere interrogato tramite delle *query* per ricavare le informazioni sullo stato del fenomeno. Ovviamente tale visione non può essere sempre valida ma, in molti casi, questo approccio facilita la scrittura delle applicazioni. Anche in questo caso il problema principale è come eseguire le interrogazioni spendendo la quantità minima possibile di energia. Il progetto più avanzato in questo ambito è forse il TinyDB, sviluppato dalla University of California at Berkeley [6].

Elaborazioni all'interno della rete

I nodi possono essere programmati per svolgere delle elaborazioni sui dati per aumentare la longevità dell'intera rete. Come accennato in precedenza le trasmissioni via radio consumano la maggior quantità di energia, quindi elaborazioni sui dati che tendano a ridurre la quantità di traffico prodotto non fanno altro che ridurre considerevolmente il consumo di energia di ogni nodo. Le maggiori ricerche in questo campo si sono focalizzate sulla compressione dei dati e principalmente sulla possibilità di aggregarli in modo da raggiungere il massimo livello di efficienza in ogni trasmissione.

Algoritmi distribuiti

Come sappiamo i nodi non hanno come unico scopo quello di acquisire informazioni dall'ambiente circostante ma possono essere muniti di attuatori,

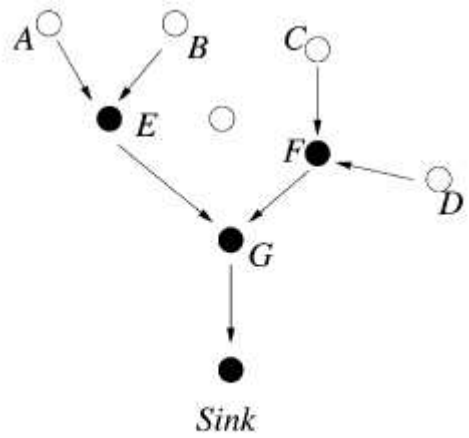

Figura 1.7: Aggregazione dei dati.

come valvole, e controllori, per raggiungere determinati scopi. Un esempio potrebbe essere l'impiego di una rete di sensori in campo automobilistico. I questo caso ogni nodo collabora con gli altri per aumentare il confort e la sicurezza nella guida. Il problema principale in questo campo è il controllo distribuito: come i nodi, sparsi nella rete, possano prendere un decisione comune quando si verifica un evento [17].

Capitolo 2

Piattaforma hardware

In questo capitolo analizzeremo le caratteristiche dei principali componenti hardware e alcune delle piattaforme per reti di sensori disponibili sul mercato.

2.1 Architettura di un singolo nodo

Tipicamente, i componenti hardware che formano un singolo nodo sensore sono: il modulo di comunicazione, dei sensori, un modulo di memoria non volatile e una batteria. Spesso comunque, le applicazioni richiedono la possibilità di aggiungere delle espansioni, come attuatori, controllori, etc. Risulta quindi utile fornire i nodi di un connettore che possa interfacciarsi con il più alto numero di dispositivi.

In questo paragrafo descriveremo le caratteristiche dei componenti piú interessanti, in relazione alle esigenze delle reti di sensori, come:

Microcontrollore

I microcontrollori che possono essere montati su un nodo sensore devono essere molto semplici e progettati per consumare poca energia, ma la caratteristica fondamentale che devono possedere è la possibilità di funzionare in diverse modalità (dalla *active* alla *sleep*). Questo permette di regolare il consumo di energia in base delle funzionalità che la rete richiede in un dato momento. Un altro aspetto da tenere

in considerazione sono le dimensioni del *chip* e la quantità di memoria disponibile all'interno di questo.

I microcontrollori più utilizzati per le reti di sensori sono l'MPS 430 della Texas Instruments o quelli della famiglia Atmel.

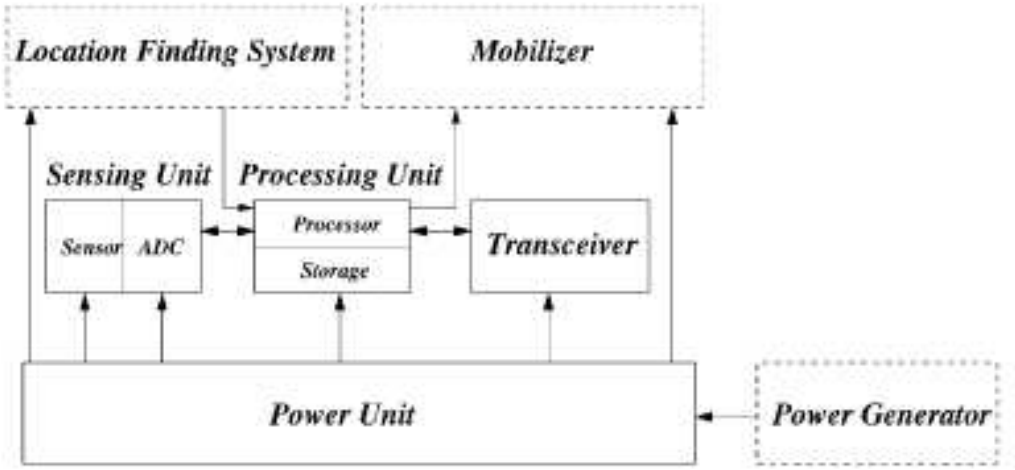

Figura 2.1: Componenti di un nodo sensore.

Sistema di comunicazione

Tipicamente viene utilizzato un sistema di comunicazione ad onde radio ma esistono particolari applicazioni per cui sono possibili altre soluzioni. Ad esempio, se la rete dovesse lavorare sott'acqua si potrebbe utilizzare un sistema di comunicazione ad ultrasuoni. Altri sistemi utilizzano dispositivi ottici per la trasmissione delle informazioni.

Nel caso di onde radio, i maggiori sforzi sono concentrati nello studio di sistemi in grado di risvegliare un nodo non appena avvertono l'inizio di una trasmissione. Questo favorisce il risparmio di energia, in quanto il maggior numero di nodi può attendere in modalità *sleep*, il verificarsi di un evento (che viene segnalato dai pochi nodi attivi con l'invio di

un pacchetto). Il risveglio può riguardare tutti i nodi vicini in ascolto oppure solo quei nodi direttamente indirizzati.

Attualmente i ricetrasmettitori radio più utilizzati sono i dispositivi RFM TR1001, Infineon o Chipcon. Generalmente viene utilizzata la modulazione ASK o FSK, solo il Berkeley PicoNodes usa la modulazione OOK.

Alimentazione

Normalmente i nodi sono alimentati da batterie esauribili ed è necessario prevedere, dove possibile, sistemi per il ricarico di energia, come celle solari, etc.

2.2 Le piattaforme

Di seguito riportiamo alcune delle piattaforme hardware maggiormente utilizzate. Queste si distinguono per la capacità di elaborazione disponibile e per le dimensioni.

2.2.1 Rockwell WINS

Questo tipo di nodo sensore è racchiuso in un contenitore delle dimensioni di 3.5" x 3.5" x 3" ed è fornito di due connettori da 40-pin, costituiti da linee dati, linee di controllo e linee di alimentazione, e sono in grado di interfacciarsi con il più largo numero di schede di sensori. Inoltre tali connettori sono in grado di comunicare con diverse interfacce come RS232, SPI e USB.

Il Rockwell WINS si distingue per la non trascurabile capacità elaborativa. Di seguito descriviamo i componenti più importanti.

Microcontrollore

Il processore montato su questa piattaforma è un Intel StrongArm SA1100. SA1100 è un microprocessore *general-purpose*, 32-bit RISC, basato sull'architettura ARM che è considerata una tra le più efficienti in termini di

Figura 2.2: Nodo sensore Rockwell WINS.

MIPS/Watt. Programmi e dati sono memorizzati in 1MB di SRAM e 4MB di memoria flash. Il processore può lavorare in tre stati: *normal*, *idle* e *sleep*.

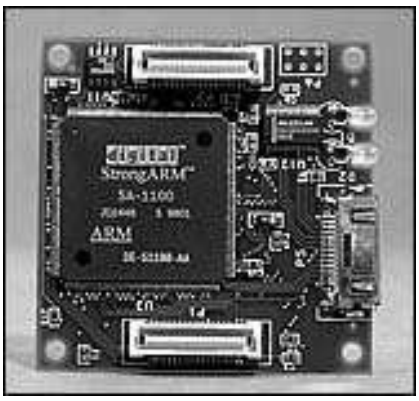

Figura 2.3: Processore SA1100

Radio

Il modulo radio utilizza un chip-set RDSSS9M Digital Cordless Telephone (DCT) della Conexant Systems, Inc. che consente una larghezza di banda di ben 900 MHz. Può lavorare in uno dei 40 canali disponibili, che possono essere selezionati per mezzo del controllore dedicato di cui è fornito. È in grado di trasmettere il segnale a differenti livelli di potenza (tra 1 e 100

mW) abilitando l'uso dell'algoritmo per l'ottimizzazione della potenza in trasmissione.

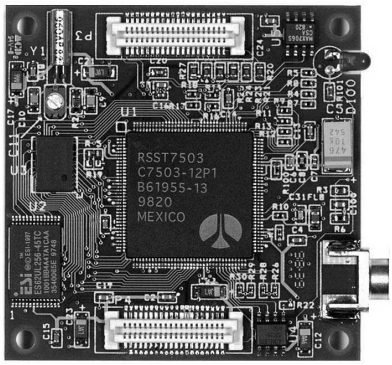

Figura 2.4: Modulo radio.

2.2.2 Mica2 e Mica2dot

I ricercatori della University of California in Berkeley hanno sviluppato una serie di nodi sensore che costituiscono la piattaforma hardware del software da noi utilizzato e perciò verranno analizzati in dettaglio. Questi nodi rappresentano la piattaforma per reti di sensori di terza generazione rispetto famiglia di nodi sviluppati dalla Berkeley University.

I nodi sensore Mica2 sono equipaggiati con 512 KB di memoria flash per memorizzare in modo permanente i dati raccolti e dispongono di un connettore a 51-pin per l'aggiunta di un espansione (attuatori, dispositivi sensori, etc.). Possiedono un clock esterno a 32 kHz che il sistema operativo, in questo caso TinyOS, utilizza per sincronizzarsi con i vicini a meno di +/- 1 ms e per assicurarsi che questi siano attivi ed in grado di ascoltare le trasmissioni. I nodi Mica2 possiedono tre led, uno rosso uno giallo e uno verde, che possono essere comandati via programma e utilizzati per visualizzare una qualche informazione di stato. Infine entrambe le piattaforme forniscono un

collegamento seriale (sullo stesso connettore a 51-pin) per i trasferimenti di dati e l'installazione delle applicazioni su ciascun nodo.

Figura 2.5: Mica2.

Figura 2.6: Mica2dot.

Microcontrollore

Il processore montato sulla piattaforma Mica è un Atmel ATMega128 AVR. AVR è un architettura Harvard a 8-bit con una memoria separata per le istruzioni e per i dati. I microcontrollori AVR supportano lo stato *active* e quello *sleep*.

Radio

I sistema di comunicazione radio utilizzato è un Chipcom CC1000 progettato per applicazioni che richiedono un basso consumo di energia ed è in grado di comunicare da pochi metri fino a centinaia di metri. Utilizza la codifica Manchester, la modulazione FSK, supporta bande di frequenza a 315, 433, 868 e 915Mhz ed è in grado di lavorare a frequenze comprese tra 300 e 1000 MHz. Il sistema radio può lavorare in quattro distinte modalità: *transmit*, *receive*, *idle* e *sleep*. Questo significa che mentre trasmette non é in grado di ascoltare eventuali trasmissioni da parte di altri nodi, quindi le collisioni, se avvengono , non sono riconosciute. In effetti il protocollo MAC utilizzato è del tipo CSMA/CA.

Nella figura 2.7 riportiamo alcune caratteristiche della piattaforma Mica2.

Component	*Rate*	*Startup time*	*Current consumption*
CPU Active	4 MHz	N/A	4.6 mA
CPU Idle	4 MHz	1 us	2.4 mA
CPU Suspend	32 kHz	4 ms	10 uA
Radio Transmit	40 kHz	30 ms	12 mA
Radio Receive	40 kHz	30 ms	3.6 mA
Photo	2000 Hz	10 ms	1.235 mA
I2C Temp	2 Hz	500 ms	0.150 mA
Pressure	10 Hz	500 ms	0.010 mA
Press Temp	10 Hz	500 ms	0.010 mA
Humidity	500 Hz	500 ms	0.775 mA
Thermopile	2000 Hz	200 ms	0.170 mA
Thermistor	2000 Hz	10 ms	0.126 mA

Figura 2.7: Alcune caratteristiche della piattaforma Mica2.

2.2.3 Spec Motes

Le dimensioni i questo tipo di nodo lo rendono quello che più si avvicina alla piattaforma del futuro perché tra tutti è quello dalle dimensioni più piccole.

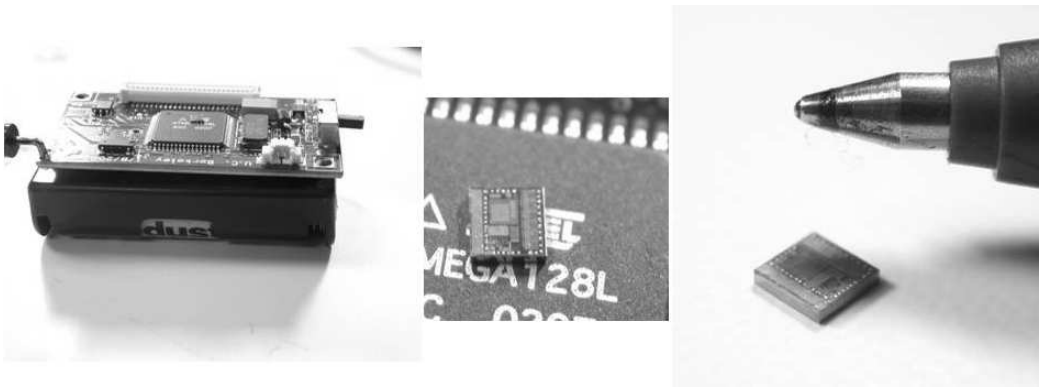

Figura 2.8: Confronto delle dimensioni di Spec Motes rispetto a Mica2.

Di seguito ne elenchiamo le principali caratteristiche:

- le dimensioni approssimative sono pari a 2 mm x 2.5 mm;
- monta un processore simile all'AVR con un cuore RISK;
- incorpora nel chip un convertitore analogico/digitale a otto canali;
- possiede un ricetrasmettitore radio Chipcon CC1000 con modulazione FSK;
- supporta la comunicazione radio criptata;
- possiede in oscillatore a 32 KHz;
- fornisce un interfaccia RS232 (UART compatibile).

Le figure 2.9 e 2.8 evidenziano alcuni particolari.

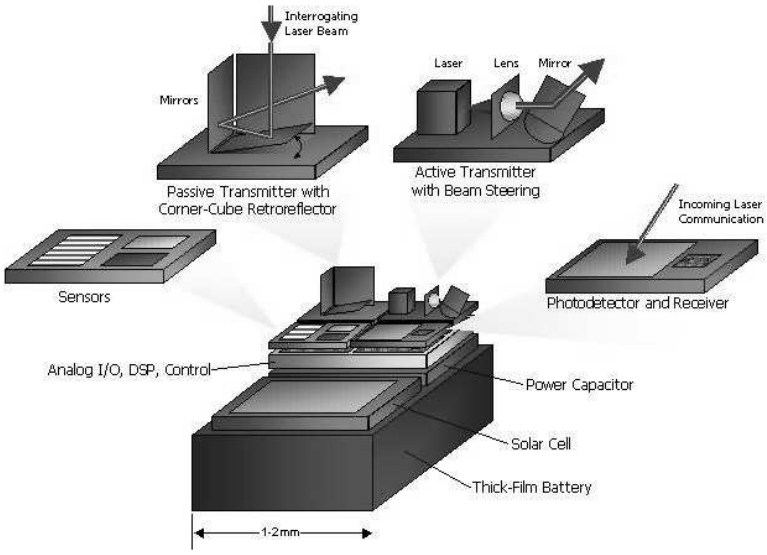

Figura 2.9: Dettaglio dei componenti di Spec Motes.

2.3 Schede di sensori

Un sensore è un dispositivo che rileva i parametri di un sistema, evitando il più possibile che questi siano disturbati dal sensore stesso. Infatti i sensori devono essere piccoli e a basso consumo di potenza per limitare gli scambi di energia con il sistema. Caratteristiche importanti per i sensori sono:

- tempo di risposta;
- risoluzione;
- efficienza (rapporto tra la potenza in ingresso e quella in uscita);
- accuratezza.

Nelle reti di sensori è poi utile tener conto di un altro parametro: il consumo di energia. Le attività che portano via la maggior quantità di energia sono:

- il campionamento del parametro e la sua conversione in un segnale elettrico
- il trattamento del segnale elettrico (es. pulizia dal rumore)
- la conversione del segnale da analogico a digitale

Comunque bisogna precisare che il fattore che influisce maggiormente sul consumo di energia è il tipo di sensore utilizzato. Ad esempio, nella misurazione della pressione o dell'umidità sono necessari diversi secondi prima di ottenere un valore, il che implica un considerevole dispendio di energia, dato che, per tutto il tempo del campionamento il sensore consuma potenza. Ciò non accade per sensori passivi come i termistori, nei quali la resistenza del conduttore varia la variare della temperatura dell'ambiente, quindi occorrono pochi millisecondi per effettuare la misurazione.

Di seguito riportiamo le caratteristiche principali delle schede di sensori per le piattaforme analizzate in precedenza.

2.3.1 Sensori per Rockwell WINS

I dispositivi sensore disponibili per tale piattaforma sono i seguenti:

Sensori acustici

La scheda dei sensori acustici impiega un piccolissimo microfono del tipo Knowles BL1785. Questo tipo di microfono presenta una frequenza di *cutoff* di 4 Hz, mentre la massima frequenza a cui possono lavorare è di 2 kHz;

Magnetometro

Attualmente è in fase di test e monta un chip tipo Honeywell HMC1001. La sensibilità di tale dispositivo è di 27 mgauss, mentre la larghezza di banda è di 10 Hz.

Accelerometro

Questa scheda include dei sensori in grado di rilevare la temperatura e la pressione ed un accelerometro ad alta velocità che campiona

ad una frequenza pari a 16 kHz. Questi ultimi sono stati costruiti principalmente per analizzare le vibrazioni dei macchinari.

Sensori sismici

Questi sensori sono stati progettati per rilevare eventi sismici a bassa frequenza. La scheda monta un chip tipo Mark IV la cui sensibilità è di 1 mg e impiega un convertitore sigma-delta tipo Analog Devices AD 7714, che produce un campione di 20 bit ad una frequenza compresa tra 1 Hz e 400 Hz.

2.3.2 Sensori per Mica2

La modularità che sta alla base del progetto di questa piattaforma ha permesso la realizzazione di numerose schede di sensori di vario tipo. Nelle figure (da inserire...) ne vengono mostrate alcune e la tabella 2.1 ne riassume le principali caratteristiche.

La scheda di riferimento per i nodi tipo Mica2 è la cosiddetta Mica Sensorboard (MTS300CA), una scheda molto flessibile dalle varie modalità di utilizzo. Questa scheda infatti può essere impiegata per applicazioni che includono il rilevamento della presenza di veicoli, l'analisi di movimenti sismici, il riconoscimento di oggetti in movimento, la localizzazione ed altre applicazioni. Il resto del paragrafo analizza i dettagli di tale scheda.

Microfono

Il microfono può avere due principali usi:

- localizzazione acustica
- registrazione e misurazione dei suoni

La localizzazione acustica è una possibilità offerta da questa scheda che può risultare molto utile. Il funzionamento è il seguente. Un nodo invia un pacchetto via radio e contemporaneamente emette un suono tramite un segnalatore (disponibile sulla scheda). Un nodo vicino, all'arrivo del pacchetto,

azzera un contatore che poi incrementa con una certa frequenza. Il contatore viene incrementato fino a quando lo stesso nodo non avverte il segnale acustico. A questo punto, moltiplicando il valore del contatore per la frequenza di incremento, si ottiene all'incirca l'intervallo di tempo occorso al segnale acustico per raggiungere il destinatario. Conoscendo quindi il valore della velocità del suono si ottiene in modo approssimato la distanza tra due nodi.

Segnalatore

Questo dispositivo è in grado di emettere un unico suono alla frequenza di 4 kHz e viene utilizzato principalmente nella localizzazione dei nodi.

Luce e temperatura

Questa scheda dispone di sensori in grado di misurare la quantità di luce nell'ambiente e la temperatura.

I sensore della luce è semplicemente una fotocellula CdSe. La massima sensibilità di questa fotocellula permette di rilevare raggi di luce fino ad una lunghezza d'onda pari a 690 nm.

Il termistore è un Panasonic ERT-J1VR103J ed è in grado di rilevare temperature che vanno da -40 a 70 C.

Accelerometro a 2-Assi

L'accelerometro è un dispositivo MEMS a 2-assi ed è in grado di rilevare accelerazioni fino a ± 2 g. È utilizzabile per rilevare movimenti, vibrazioni e/o misurazioni sismiche.

Magnetometro a 2-Assi

I sensore è un Honeywell HNC1002. Il magnetometro può misurare il campo magnetico terrestre ed altri piccoli campi. Un applicazione in cui può essere impiegato è il rilevamento della presenza di veicoli. In un esperimento è stato possibile rilevare i disturbi prodotti da un automobile nel raggio di 4,57

metri.

Tipo	Descrizione
MTS101	monta un termistore di precisione e sensori di luminosità, ed è utilizzabile in varie aree
MTS300/MTS310	supporta una grande varietà di sensori per MICA e MICA2
MDA500	fornisce una flessibile interfaccia per collegamenti con i nodi tipo MICA2DOT
MTS400/420	è in grado di effettuare il monitoraggio ambientale ed eventualmente può essere montato un GPS (MICA2)
MDA300	è in grado di acquisire dati e monitorate l'ambiente (MICA2)
MTS510	supporta sensori di luce, accelerazione ed un microfono (MICA2DOT)
MEP401	moduli per luce, umidità, pressione barometrica e temperatura (MICA2)
MEP500	modulo per temperatura e umidità (MICA2DOT)

Tabella 2.1: Schede disponibili per la piattaforma Mica

Capitolo 3

TinyOS

In questo capitolo descriveremo il sistema operativo TinyOS, sul quale si basa il simulatore Tossim. Inoltre spiegheremo come sviluppare un applicazione per reti di sensori utilizzando gli strumenti forniti dal sistema operativo.

3.1 I sistemi operativi e le reti di sensori

Questo paragrafo cerca di mettere in luce le caratteristiche delle reti di sensori in relazione alla progettazione di un sistema operativo.

Uno degli aspetti più evidenti delle reti di sensori sono le ridotte dimensioni della piattaforma hardware e la scarsa disponibilità di energia. Questi fattori influiscono sulla capacità di elaborazione del sistema, sulla quantità di informazioni che è possibile scambiare e sulla capacità di comunicare con l'esterno. Il software sviluppato per le reti di sensori dovrà essere perciò molto semplice, per evitare operazioni inutili, e poco ingombrante.

Le reti di sensori sono soggette a continue sollecitazioni. Si pensi ad esempio alla quantità di pacchetti che ogni nodo deve trattare, sia perché ha la necessità di inviare le informazioni raccolte, sia perché la rete comunica secondo un protocollo *multi-hop*. Inoltre l'ambiente esterno deve essere continuamente monitorato e la rete, per poter garantite i servizi essenziali, deve continuamente scambiarsi delle informazioni di "servizio" (si pensi al

problema della mobilità o a quello della sincronizzazione temporale). In pratica il sistema deve gestire una grande quantità di eventi nel modo più veloce possibile, per evitare di perdere informazioni preziose.

Il livello di parallelismo insito in questo tipo di sistemi è molto basso rispetto ai sistemi convenzionali. Questo è dovuto principalmente alle limitazioni fisiche dei nodi (bassa capacità di elaborazione, poca memoria, etc.) quindi si preferisce sfruttare le poche risorse per effettuare le operazioni essenziali. Addirittura, i dispositivi mettono direttamente a disposizione dei programmi le interfacce dei loro controllori, senza perciò l'utilizzo di intermediari, come controllori collegati a bus sofisticati, dispositivi che forniscono un interfaccia ad alto livello dell'hardware, ed altri, che aumentano il livello di parallelismo senza poi, in questo caso, apportare degli effettivi benefici.

Come abbiamo gia visto, le reti di sensori possono essere utilizzate in numerose applicazioni, differenti tra loro. Queste richiedono in molti casi dell'hardware dedicato e quindi sono necessarie delle reimplementazioni dello stesso software per adeguarlo ai diversi dispositivi. Tuttavia, se il software viene sviluppato seguendo il criterio di modularità, non sono necessarie complete reimplementazioni di un'applicazione ma dovrebbe essere sufficiente sostituire alcuni moduli per ottenere il medesimo risultato. Lo stesso problema sorge quando ad esempio si vorrebbe sostituire un protocollo all'interno di un applicazione. In generale quindi, si dovrebbe progettare un sistema operativo in grado di adattarsi facilmente a tutti i cambiamenti tecnologici, sia hardware che software.

I nodi in alcune applicazioni si trovano a lavorare in ambienti critici che ne potrebbero alterare il funzionamento. Questi possono guastarsi o rimanere bloccati quindi è necessario sviluppare applicazioni tolleranti ai guasti. Anche il sistema operativo deve fare la sua parte, cioè deve supportare eventuali guasti dei dispositivi e favorire lo sviluppo di applicazioni affidabili e distribuite, in conclusione deve garantire operazioni robuste.

3.2 Introduzione a TinyOS

TinyOS è un sistema operativo *open-source*, sviluppato dalla University of California at Berkeley. Data la possibilità di modificare il codice, questo sistema operativo è diventato la piattaforma di sviluppo per ogni soluzione proposta nel campo delle reti di sensori. In effetti grossi contributi sono stati forniti dalla comunità di sviluppatori, lo testimonia la lunga lista di progetti attivi relativi a tutti campi della ricerca, da protocolli per l'instradamento dei pacchetti alla localizzazione, dalla realizzazione di un interfaccia grafica per lo sviluppo delle applicazioni all'estensione del compilatore `ncc` per il supporto di nuove piattaforme hardware (come la piattaforma 8051). Comunque per avere un idea delle attività portate avanti basta esplorare la cartella /tinyos-1.x/contrib.

TinyOS è stato progettato principalmente per le reti di sensori. A differenza delle tradizionali architetture hardware, dove disponiamo di grandi quantità di memoria, complessi sottosistemi per la gestione dei dati di ingresso e per quelli d'uscita, forti capacità di elaborazione e sorgenti di energia praticamente illimitate, nelle reti di sensori ci troviamo a confronto con sistemi di piccole dimensioni, fonti di energia limitate, scarsa quantità di memoria, modeste capacità di elaborazione, etc. Sono necessarie quindi soluzioni molto semplici ed efficienti, e che soprattutto riducano la massimo i consumi di energia. Anche il sistema operativo risente di queste limitazioni. Infatti TinyOS non possiede un nucleo ma permette l'accesso diretto all'hardware, inoltre sparisce i concetto di processore virtuale per evitare i cambi di contesto, e quello di memoria virtuale: la memoria viene infatti considerata come un unico e lineare spazio fisico, che viene assegnato alle applicazioni a tempo di compilazione. In pratica viene eliminato qualsiasi tipo di *overhead*, poiché nelle reti di sensori questo causa un inutile dispersione di energia senza portare tangibili vantaggi.

Queste scelte impattano direttamente sull'architettura che risulta molto compatta e ben si sposa con le dimensioni ridotte della memoria che a volte raggiunge la quantità di pochi chilobyte.

Un tipico *layout* di un programma scritto per TinyOS è mostrato in figura 3.1.

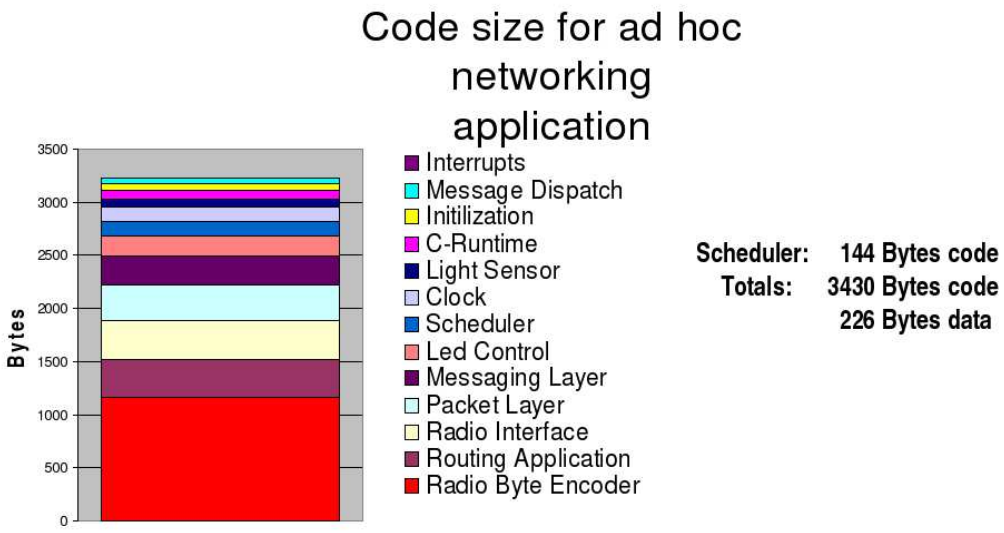

Figura 3.1: Il layout tipico di un programma TinyOS.

Lo scopo dichiarato dei progettisti di TinyOS era infatti quello di

- ridurre i consumi di energia;
- ridurre il carico computazionale e le dimensioni del sistema operativo;
- supportare intensive richieste di operazioni che devono essere svolte concorremente e in maniera tale da raggiungere un alto livello robustezza ed un efficiente modularità.

Per soddisfare il requisito della modularità, TinyOS favorisce lo sviluppo di una serie di piccoli componenti, ognuno con un ben precisa funzione, che realizza un qualche aspetto dell'hardware del sistema o di un'applicazione. Ogni componente poi, definisce un interfaccia che garantisce la riusabilità del componente ed eventualmente la sua sostituzione. Guardando in modello a strati del sistema operativo (figura 3.2), possiamo intuire come sia facilitato lo sviluppo di applicazioni mediante il riutilizzo dei componenti esistenti.

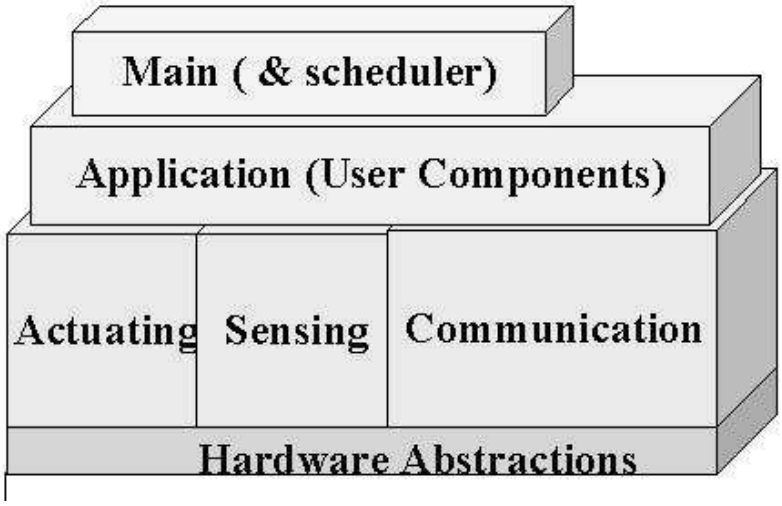

Figura 3.2: Modello a strati di TinyOS.

TinyOS è stato implementato seguendo il modello ad eventi. Nei tradizionali modelli infatti, è necessario allocare un certa quantità di memoria sullo *stack* per ogni attività in esecuzione e inoltre, il sistema è soggetto a frequenti commutazioni di contesto per servire ogni tipo di richiesta, come l'invio di pacchetti, la lettura di un dato su un sensore, etc. Naturalmente i nodi non dispongono di grosse quantità di memoria ed il modello ad eventi ben si addice ad un sistema continuamente sollecitato.

Nel campo delle reti di sensori il consumo di energia è un fattore critico e l'approccio basato sugli eventi utilizza il microprocessore nella maniera più efficiente possibile. Quando il sistema viene sollecitato da un evento questo viene gestito immediatamente e rapidamente. In effetti non sono permesse condizioni di bloccaggio né attese attive che sprecherebbero energia inutilmente. Quando invece non ci sono attività da eseguire il sistema mette a riposo il microprocessore che viene risvegliato all'arrivo di un evento.

3.3 TinyOS in dettaglio

TinyOS è un insieme di librerie e applicazioni che possono essere riutilizzate per costruire nuove applicazioni. Le librerie includono i protocolli di rete, i servizi distribuiti, i gestori dei sensori, e degli strumenti per l'acquisizione dei dati. A loro volta le librerie e le applicazioni sono costituite da un insieme di componenti i quali rappresentano i mattoni con cui si costruisce un applicazione.

Un altro elemento importante dell'architettura di TinyOS è lo schedulatore. Come abbiamo già discusso, TinyOS utilizza un modello di programmazione basato su macchine a stati invece del tradizionale modello basato su *thread*. Ogni componente transita da uno stato ad un altro mediante una richiesta di operazioni oppure in seguito all'arrivo di un evento. Ogni transizione avviene praticamente in modo istantaneo, cioè non esiste nessun evento che possa interromperla, e richiede pochissime operazioni da parte del processore. Comunque per operazioni che richiedono lunghe elaborazioni sono previsti speciali meccanismi.

Un altro vantaggio nella scelta di tale modello di programmazione consiste nel fatto che il livello di astrazione prodotto dall'hardware si propaga direttamente nel software, così segnali di interruzione provenienti dall'hardware si traducono immediatamente in eventi software.

Di seguito riportiamo un analisi più dettagliata sia dei componenti sia del modello si esecuzione.

3.3.1 I componenti

Ogni componente è un unità indipendente e svolge un determinato compito. L'insieme di componenti che forma un applicazione è infatti costituito da componenti preesistenti (che appartengono a librerie o applicazioni sviluppate in precedenza) e da nuovi componenti, cioè quelli necessari a completare l'applicazione e viene chiamato grafo dei componenti. Il grafo fissa una gerarchia nel senso che il processo di composizione dell'applicazione produce

una stratificazione all'interno della stessa. I componenti di livello inferiore segnalano eventi ai quelli di livello più alto, mentre i componenti di livello superiore richiedono dei servizi ai componenti di livello più basso. Lo strato inferiore infine, è formato da componenti che rappresentano direttamente i dispositivi hardware.

Un componente è formato da quattro parti tra loro collegate:

- un insieme di comandi;
- un insieme di eventi;
- un *frame*;
- un certo numero di *task*.

Comandi, eventi e *task* operano all'interno del frame, modificandone lo stato.

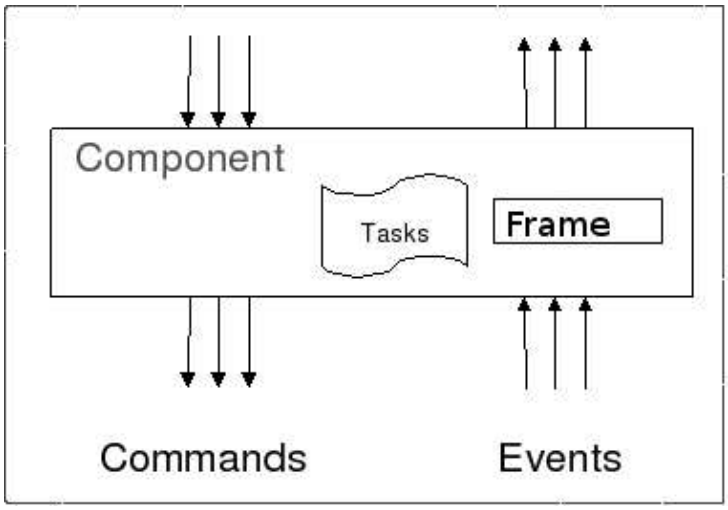

Figura 3.3: Le parti di un componente.

Un componente definisce poi delle interfacce che permettono di realizzare applicazioni modulari.

3.3.2 Il *Frame*

Il *frame* viene allocato a tempo di compilazione e permette di conoscere non solo la quantità di memoria richiesta da un componente ma quella dell'intera applicazione. Questa soluzione permette di evitare gli sprechi dovuti all'*overhead* associato alle allocazioni dinamiche. Infatti il sistema operativo non deve compiere delle operazioni per risolvere gli indirizzi in memoria, dato che i riferimenti all'interno del programma corrispondono ad indirizzi fisici.

3.3.3 I Comandi

I comandi sono richieste di un servizio fornito da un componente di livello più basso e non sono bloccanti. Generalmente un comando deposita dei parametri nel *frame* e poi attiva un *task*, ma è possibile anche che questo chiami un altro comando. In quest'ultimo caso, il componente non può attendere per un tempo indeterminato la fine della chiamata. Il comando, ritorna poi un valore che indica se la chiamata ha avuto successo o meno.

3.3.4 Gli Eventi

Gli eventi sono, direttamente o indirettamente, dei gestori delle interruzioni hardware. Il componente di livello più basso trasforma un interruzione hardware in un evento che può essere provocato da un interruzione esterna, da timer, o dal contatore, e poi propaga tale richiesta ai livelli più alti. Similmente ai comandi, un evento può depositare dei parametri nel frame e poi attivare un *task*. Un evento può richiamare altri eventi e alla fine chiamare un comando, come a formare una catena che prima sale e poi scende. Per evitare che questa catena si possa chiudere viene impedito ai comandi di generare eventi.

3.3.5 Modello di concorrenza: i *task*

TinyOS esegue un solo programma alla volta. Questo è composto da un solo tipo di attività indipendente: il *task*. I task non hanno diritto di prelazione su nessuna delle attività in corso, in pratica sono atomici rispetto agli altri *task*. Inoltre possono richiamare comandi, generare eventi o attivare altri *task* all'interno dello stesso componente. La caratteristica di essere eseguito fino al suo completamento infine. permette di gestire un singolo *stack* che viene assegnato, a turno, al *task* in esecuzione.

Generalmente la scelte del *task* da mandare in esecuzione viene effettuata secondo il criterio FIFO, ma ciò non toglie che si possa modificare lo schedulatore affinché realizzi una più sofisticata scelta. Se la coda dei *task* è vuota lo schedulatore mette a riposo il microprocessore come riporta i seguente frammento di codice estratto dallo schedulatore.

```
main{
  ....
  while(1){
    while(more_tasks)
      schedule_task;

    sleep;
  }
}
```

I gestori possono sempre interrompere un task o un gestore stesso nel caso questo abbia priorità maggiore dell'evento attualmente in esecuzione.

La figura 3.4 riassume il modello di concorrenza del TinyOS.

3.4 NesC

NesC è un linguaggio di programmazione ideato per lo sviluppo su sistemi embedded. NesC ha una sintassi simile al C e supporta il modello di

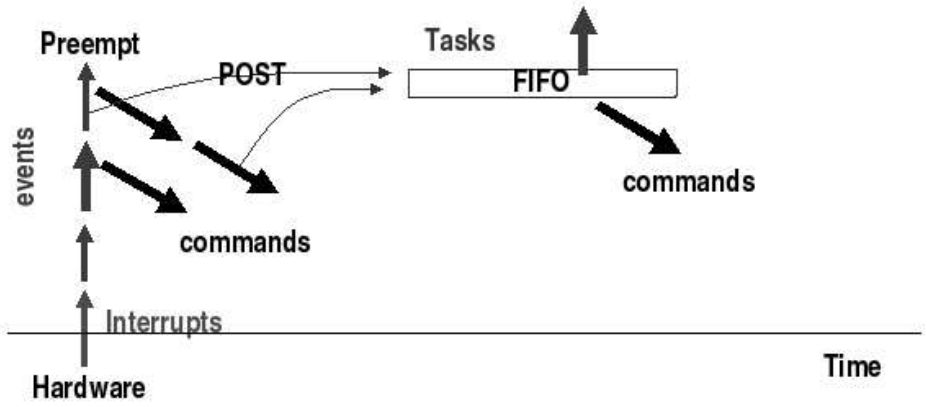

Figura 3.4: Modello di concorrenza di TinyOS.

concorrenza del TinyOS.

Le caratteristiche principali del linguaggio nesC sono:

- *Separazione della costruzione e della composizione:* sviluppare un'applicazione in nesC significa realizzare una serie di componenti che verranno poi assemblati per produrre un codice eseguibile. Ogni componente ha due obbiettivi da realizzare: deve definire le specifiche del suo comportamento e deve implementare tali specifiche. I componenti presentano un modello di concorrenza interno nella forma dei *task* e il controllo del processore passa da un componente ad un altro in seguito alle richieste di un servizio e agli eventi che sopraggiungono.

- *Specificare il comportamento del componente mediante una serie di interfacce:* le interfacce possono essere fornite o usate dai componenti. Le interfacce fornite rappresentano le funzionalità che un componente intende mettere a disposizione degli altri. Le interfacce usate invece, sono le funzionalità di un altro componente che si intendono utilizzare per realizzare il proprio comportamento;

- *Le interfacce sono bidirezionali:* queste specificano una serie di funzioni che devono essere implementate dal fornitore dell'interfaccia e un

altro insieme di funzioni che devono essere implementate dal componente che le utilizza. Questo permette di realizzare operazioni complesse con un singola interfaccia. Ciò è necessario in quanto l'esecuzione di un comando non può essere un operazione bloccante, anche se esistono delle operazioni che richiedono un tempo non determinato per giungere a conclusione, come ad esempio l'invio di un pacchetto. In questo caso viene associato al comando un evento che segnala il termine dell'operazione. Quindi se si utilizza un'interfaccia per l'invio dei pacchetti, un componente non può chiamare il comando `send` (invio del pacchetto), senza implementare l'evento `sendDone` (invio del pacchetto completato);

- *I componenti sono collegati staticamente attraverso le loro interfacce:* questo per raggiungere una maggiore velocità di esecuzione, per incoraggiare le implementazioni robuste, e per permettere l'analisi statica del programma. Nel caso si annidassero possibili corse critiche, il compilatore è in grado di accorgersene e di segnalarlo all'utente;

- *Il modello di concorrenza del nesC aderisce a modello del TinyOS:* il modello del compilatore nesC è basato su *task*, che sono eseguiti fino al loro completamento e da gestori delle interruzioni, che possono interrompere i *task* e, se di priorità maggiore, anche i gestori stessi.

3.4.1 Le interfacce

Ogni componente definisce le interfacce fornite e quelle usate che rappresentano gli unici punti di accesso per interagire con lo stesso. Un interfaccia dichiara una serie di funzioni che possono essere di due tipi: comandi ed eventi. I comandi devono essere implementati dal componente che li fornisce mentre gli eventi devono essere implementati dal componente che li usa. Un comando generalmente è una richiesta di servizio, mentre l'evento segnala il completamento di un servizio. Gi eventi possono essere generati anche in modo asincrono, per esempio in seguito ad un interruzione hardware o al-

l'arrivo di un pacchetto. Si possono usare o fornire più di un interfaccia e istanze multiple di una stessa interfaccia (ovviamente rinominandole).

Di seguito riportiamo il codice di una semplice interfaccia, implementata nel file /tinyos-1.x/tos/interfaces/SendMsg.nc :

```
interface SendMsg
{
  command result_t send(uint16_t address, uint8_t length,
                        TOS_MsgPtr msg);

  event result_t sendDone(TOS_MsgPtr msg,
                          result_t success);
}
```

3.4.2 I componenti

Un componente può esser di due tipi: modulo o configurazione.

I moduli implementano le interfacce che dichiarano (sia i comandi "esportati" che gli eventi "importati"). Ecco qui un esempio:

```
module ComponentModule {
  provides interface X
  provides interface Y

  uses interface Z
}

implementation {
 // Codice dei comandi esportati dalle interfacce X e Y
 ...

 // Codice degli eventi importati dall'interfaccia Z
 ...
```

```
}
```

Le configurazioni invece definiscono come più componenti devono essere assemblati e possono fornire delle interfacce anche se non vengono implementate esplicitamente (in effetti queste sono implementate in modo implicito dai componenti a cui sono collegate).

```
configuration ComponentConfiguration {
  provides interface X
  provides interface Y

  uses interface Z
}

implementation {
 // Codice che definisce come le interfacce X e Y si
 // collegano con l'interfaccia Z
 ...
}
```

3.4.3 Assemblaggio dei componenti

Per realizzare un applicazione è necessario collegare i vari componenti. Questa operazione viene definita "wiring" e produce il cosiddetto grafo dei componenti. In figura 3.5 viene mostrato un esempio che ora analizzeremo in dettaglio.

I componenti:

- `ComponentD`,
- `ComponentF`,
- `Application`,

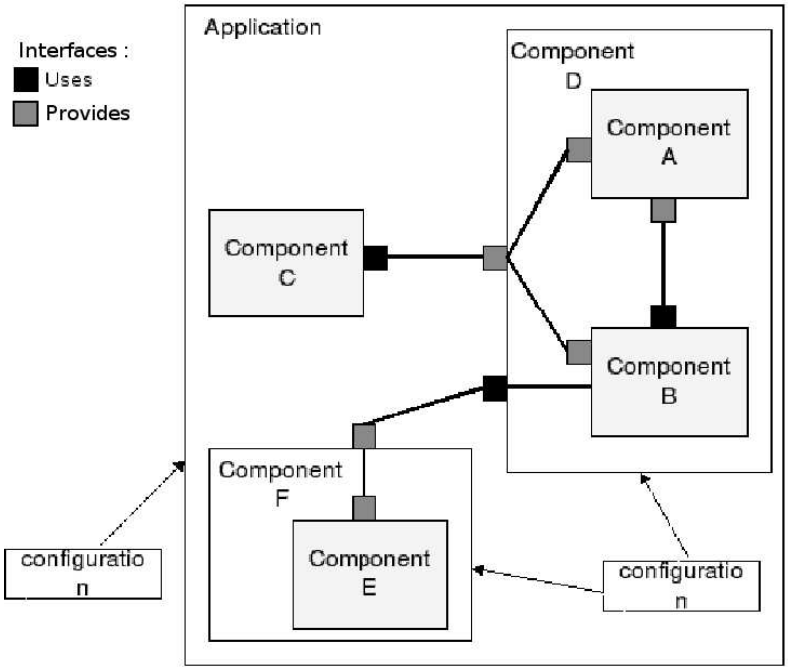

Figura 3.5: Grafo dei componenti.

sono delle configurazioni e, come abbiamo già spiegato, servono a collegare tra loro componenti preesistenti. Il resto dei componenti sono dei moduli.

Il componente `ComponentD` fornisce un interfaccia che verrà usata da `ComponentC` e utilizza un interfaccia sarà fornita da `ComponentF`

Il componente `ComponentF` fornisce una sola interfaccia che corrisponde a quella fornita da `ComponentE`

Il componente `Application` costituisce l'applicazione vera e propria. Non utilizza ne fornisce interfacce ma effettua il semplice collegamento tra i componenti `ComponentC`, `ComponentD` e `ComponentF`

Come possiamo notare la ragione per cui un componente si distingue in moduli e configurazioni è per favorire la modularità di un applicazione. Ciò permette allo sviluppatore di assemblare velocemente le applicazioni. In effetti, si potrebbe realizzare un applicazione solo scrivendo un componente di configurazione che assembli componenti già esistenti. D'altra parte questo modello incoraggia l'aggiunta di librerie di componenti tali da imple-

mentare algoritmi e protocolli che possano essere utilizzati in una qualsiasi applicazione

3.5 Stratificazione e componenti

La figura 3.6 mostra il grafo di una applicazione tipo, che potrebbe ad esempio, leggere i valori di luminosità e temperatura dell'ambiente, con una certa frequenza, e comunicare i dati raccolti via radio.

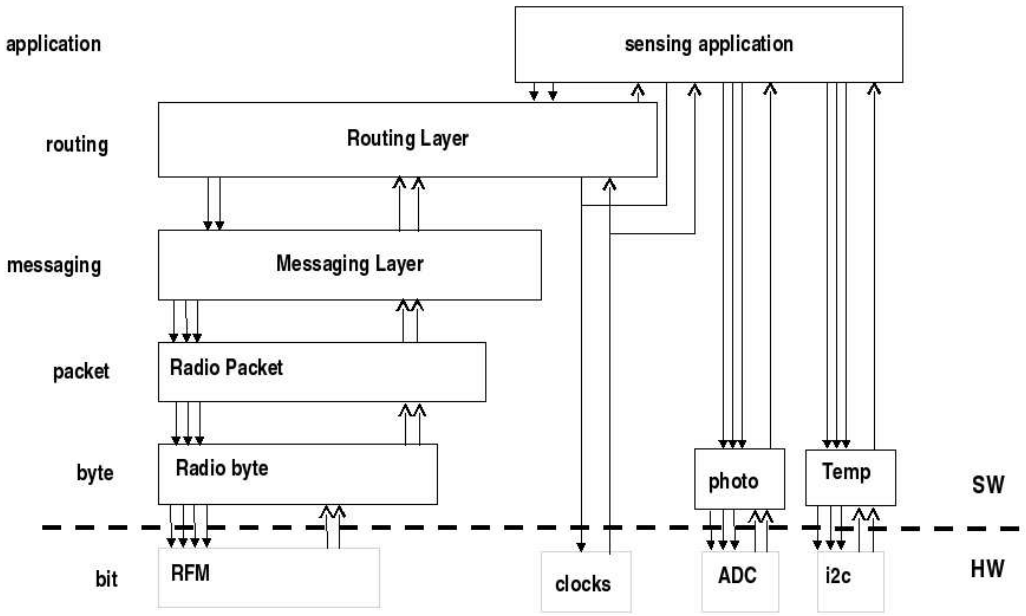

Figura 3.6: Relazione tra componenti e stratificazione di TinyOS.

Quello che ci interessa maggiormente però è mostrare le relazioni che intercorrono tra i vari componenti e la stratificazione de sistema operativo TinyOS (vedi figura 3.2). Questa relazione favorisce la modularità delle applicazioni, e come possiamo notare, se volessimo modificare il protocollo di *routing* basterebbe sostituire solo quei componenti che appartengono al blocco *Routing Layer*, lasciando inalterato il resto dell'applicazione. Lo stesso discorso vale nel caso volessimo installare la nostra applicazione su

una piattaforma diversa. I componenti di più basso livello sono direttamente collegati all'hardware, infatti ad esempio, un'interruzione hardware viene tradotta direttamente in un evento. In pratica questi componenti realizzano un astrazione software dei dispositivi fisici ai quali sono connessi. Se volessimo, in questo caso, utilizzare un sistema di comunicazione diverso basterebbe sostituire i componenti del blocco *RFM*. Ed è appunto questa l'operazione che il compilatore `ncc` esegue quando compiliamo un applicazione per una piattaforma diversa, sia questa Mica, Mica2, Mica2dot oppure PC.

3.6 Esempio di applicazione

In questo paragrafo esamineremo una semplice applicazione TinyOS denominata "Blink", che è possibile trovare nella cartella delle applicazioni /tinyos-1.x/apps/. Questa applicazione produce semplicemente l'accensione e lo spegnimento del led rosso alla frequenza di un hertz.

Blink è composta da due componenti: un modulo denominato BlinkM.nc e una configurazione chiamata Blink.nc. Possiamo subito notare che i componenti hanno nomi simili, infatti coi nomi si cerca di distinguere il tipo di componente. Ad esempio, esistono tre componenti con nomi simili: Timer, TimerC e TimerM, La "M" sta per *Module* mentre la "C" sta per *Configuration*. I particolare la "M" viene utilizzata quando un componente possiede sia un modulo che la configurazione, mentre la "C" viene utilizzata per distinguere tra un interfaccia e il componente che la fornisce. Comunque tutte le applicazioni richiedono un singolo componente di configurazione al livello più alto, che convenzionalmente è chiamato come l'applicazione stessa. Da segnalare infine che tutti i file nesC hanno l'estensione .nc.

Come possiamo notare dalla figura 3.7, il componente di configurazione `Blink` assembla quattro componenti: `Main`, `BlinkM`, `LedsC`, `ClockC`. Come detto precedentemente, lo scopo principale di un componente di configurazione è quello di favorire la modularità di un applicazione TinyOS, infatti il

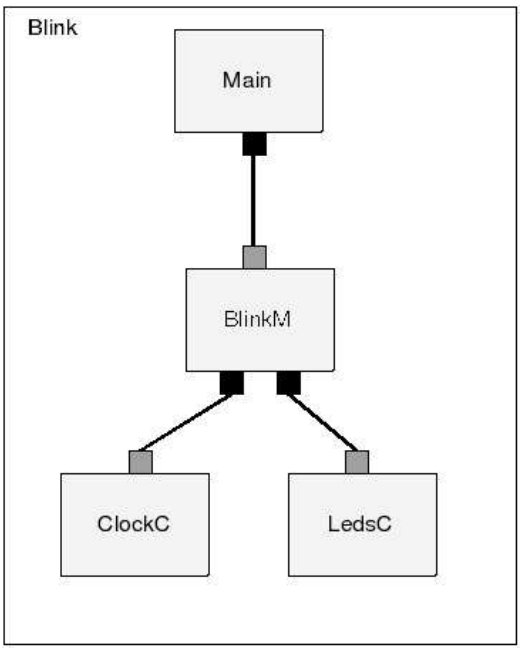

Figura 3.7: Grafo dei componenti per l'applicazione Blink.

codice in più che è stato scritto è quello per realizzare il modulo `BlinkM` e ovviamente la configurazione `Blink`.

Analizziamo ora il codice dell'applicazione a partire da Blink.nc:

```
configuration Blink {
}

implementation {
    components Main, BlinkM, SingleTimer, LedsC;
    Main.StdControl -> SingleTimer.StdControl;
    Main.StdControl -> BlinkM.StdControl;
    BlinkM.Timer -> SingleTimer.Timer;
    BlinkM.Leds -> LedsC;
}
```

Le prime due righe:

```
configuration Blink {
}
...
```

semplicemente ci dicono che questo è un componente di configurazione, chiamato `Blink`, e che non fornisce ne usa alcuna interfaccia. Questo è comune per i componenti che realizzano le applicazioni, in quanto non avrebbe senso che un applicazione fornisca un interfaccia o usi un componente.

Le due righe seguenti:

```
...
implementation {
    components Main, BlinkM, SingleTimer, LedsC;
...
```

invece, iniziano con la descrizione dell'implementazione di `Blink` e ci informano che saranno utilizzati i componenti `Main`, `BlinkM`, `LedsC` e `ClockC`.

`Main` è il componente principale di un applicazione infatti è quello che viene eseguito per primo. In particolare, prima viene eseguito il comando `init()` dell'interfaccia `StdControl` (`Main.StdControl.init()`), e poi il comando `start()` della stessa interfaccia (`Main.StdControl.start()`).

Seguono altre due linee:

```
...
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
...
```

che collegano l'interfaccia `Main.StdControl` con `SingleTimer.StdControl` e l'interfaccia `Main.StdControl` con `BlinkM.StdControl`. Questo ci dice che quando verrà chiamato il comando `Main.StdControl.init()` verranno anche chiamati i comandi `SingleTimer.StdControl.init()` e `BlinkM.StdControl.init()`. La stessa cosa accade per `Main.StdControl.start()` e `Main.StdControl.stop()` che compongono l'interfaccia `StdControl`.

Come possiamo notare `StdControl` è un interfaccia comune a molti componenti e la sua definizione, che possiamo trovare nel file /tinyos-1.x/tos/interfaces/StdControl.nc è la seguente:

```
interface StdControl
{
  /**
   * Initialize the component and its subcomponents.
   */
  command result_t init();

  /**
   * Start the component and its subcomponents.
   */
  command result_t start();

  /**
   * Stop the component and pertinent subcomponents
   * (not all subcomponents may be turned off due to
   *  wakeup timers, etc.).
   */
  command result_t stop();
}
```

Infatti, questa è l'interfaccia standard per l'inizializzazione dei componenti, per il loro riavvio e sospensione. Tipicamente il comando `stop()` viene chiamato quando si vuole mettere in sospensione (da *active* a *sleep*) un componente (e dei suoi subcomponenti), mentre il comando `start()` viene utilizzato al contrario, per la riattivazione o per il primo avvio del componente (e di alcuni dei suoi subcomponenti).

Infine le ultime righe:

```
...
```

```
    BlinkM.Timer -> SingleTimer.Timer;
    BlinkM.Leds -> LedsC;
}
```

definiscono il collegamento tra `BlinkM.Timer` con `SingleTimer.Timer` e `BlinkM.Leds` con `LedsC.Leds`

È utile a questo punto conoscere le implementazioni delle interfacce `Leds` e `Timer`.

L'interfaccia `Timer` la troviamo nel file /tinyos-1.x/tos/interfaces/Timer.nc:

```
includes Timer;

interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();

  event result_t fired();
}
```

Questa è una tipica interfaccia bidirezionale poiché oltre ai comandi viene definito un evento. I componenti che utilizzano tale interfaccia devono implementare l'evento.

L'interfaccia `Leds` la troviamo nel file /tinyos-1.x/tos/interfaces/Leds.nc:

```
interface Leds {
  async command result_t init();

  async command result_t redOn();
  async command result_t redOff();
  async command result_t redToggle();

  async command result_t greenOn();
  async command result_t greenOff();
```

```
    async command result_t greenToggle();

    async command result_t yellowOn();
    async command result_t yellowOff();
    async command result_t yellowToggle();

    async command uint8_t get();

    async command result_t set(uint8_t value);
  }
```

L'attributo `async` applicato ai comandi ci dice che questi possono essere richiamati all'interno di eventi.

Passiamo ora al modulo `BlinkM`. Il modulo inizia con le dichiarazioni delle interfacce:

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
```

viene fornita l'interfaccia `StdControl` e usate le interfacce `Leds` e `Timer`. Questo significa che `BlinkM` deve implementare i comandi `StdControl.init()`, `StdControl.start()` e `StdControl.stop()`. Mentre deve implementare l'unico evento `Timer.fired()`.

Infine l'implementazione del modulo `BlinkM` e:

```
implementation {
```

```
    command result_t StdControl.init() {
      call Leds.init();
      return SUCCESS;
    }

    command result_t StdControl.start() {
      return call Timer.start(TIMER_REPEAT, 1000);
    }

    command result_t StdControl.stop() {
      return call Timer.stop();
    }

    event result_t Timer.fired()
    {
      call Leds.redToggle();
      return SUCCESS;
    }

  }
```

Notiamo come il comando `StdControl.start()`, quando richiamato, avvia il timer per generare, ogni mille millisecondi, un interruzione che viene trasformata in un evento che si propaga dai componenti di basso livello fino al modulo `BlinkM`. Quando il timer genera l'interruzione viene eseguito l'evento `Timer.fired()` che richiama il comando `Leds.redToggle()`, il quale modifica lo stato del led rosso.

Capitolo 4

Tossim

In questo capitolo analizzeremo le caratteristiche principali dei Tossim partendo dalla descrizione dei requisiti che dovrebbe soddisfare un qualsiasi simulatore. Per quanto riguarda Tossim, vedremo quali aspetti della realtà simulata è in grado di riprodurre, l'architettura e gli strumenti connessi a tale simulatore.

4.1 I simulatori

Il principale scopo di un simulatore è quello di fornire un ambiente controllato e il più fedele possibile della realtà che si intende analizzare. In particolare, le simulazioni forniscono un metodo rapido per testare e validare una soluzione, per confrontare le le alternative proposte o per analizzare quelle interazioni che sono difficilmente osservabili nel sistema reale. Per raggiungere il massimo grado di accuratezza è necessario considerare il comportamento del sistema a tutti i livelli. Non è sufficiente cioè, poter simulare algoritmi e protocolli di rete ma, data la complessità delle interazioni a cui sono soggette le reti di sensori, applicazioni complete. In una rete di sensori infatti, i protocolli e le applicazioni possono interagire tra loro, facendo così scomparire il concetto di stratificazione [35]. Ad esempio, nel processo di aggregazione dei dati, i protocolli di rete dipendono temporalmente dalle letture effettuate

dai sensori. Infine, come discusso precedentemente, un fattore critico che caratterizza ogni soluzione studiata per una rete di sensori è la scalabilità: un simulatore dovrebbe esser in grado di supportare reti di migliaia di nodi senza che le prestazioni decadino in modo eccessivo.

Molte ricerche condotte in questo campo hanno concluso che realizzare uno strumento sulla base dei requisiti sopra citati (fedeltà e scalabilità) è praticamente impossibile, per questo motivo sono stati sviluppati simulatori che invece di riprodurre una qualsiasi implementazione di una rete di sensori descrivono un modello dei nodi. Tra tutti citiamo ns-2 e SENSE. Questo tipo di approccio è in gradi di riprodurre i ritardi della della rete, le capacità di trasferimento, le collisioni dei pacchetti, e gli effetti prodotti dall'utilizzo di meccanismi per la gestione del consumo energetico. Tuttavia, un motivo per cui quest'approccio è risultato poco efficiente, (oltre ad non essere in grado di riprodurre il comportamento di un applicazione) è la differenza tra l'implementazione di un algoritmo fatta per essere testata dal simulatore e quella realizzata per essere installata sui nodi veri e propri. Questo causa due problemi: il primo è che, se le due implementazioni sono estremamente diverse i risultati ottenuti dalla simulazione possono non essere completamente esatti; il secondo si riferisce ad una questione più pratica: bisogna implementare due volte la stessa soluzione.

Tossim fornisce un ambiente di simulazione per reti di sensori aderente al sistema operativo TinyOS ed è stato progettato sulla base dei criteri sopra esposti, per questi motivi è ampiamente utilizzato dalla comunitàà di ricercatori e rappresenta un termine di paragone per lo sviluppo di nuovi simulatori.

4.2 Tossim

Tossim è un simulatore ad eventi discreti per applicazioni basate sul sistema operativo TinyOS. Il principale obbiettivo di Tossim è quello di simulare nel modo più fedele possibile le applicazioni realizzate per TinyOS. Infatti

permette all'utente di testare e analizzare un applicazione, ed elminarne gli errori, eseguendola in ambiente controllato e ripetibile. Tossim comunque non è la soluzione per tutti i problemi in quanto non tiene conto di alcuni aspetti del mondo reale. Per chiarire quali sono le possibilità offerte da questo strumento ne analizzeremo le caratteristiche.

4.2.1 Modello di esecuzione

Il cuore del modello di esecuzione di Tossim è la coda degli eventi. Gli *eventi Tossim* (che sono diversi dagli *eventi TinyOS*) sono generati dal sistema (per scandire il trascorrere del tempo, in fase di inizializzazione, etc.), dai comandi ed agli eventi TinyOS. Ad ogni evento è associato l'identificativo di un nodo, che permette di associare l'azione corrispondente ad un nodo, l'istante temporale in cui l'evento dovrà essere eseguito ed una funzione che rappresenta il gestore dell'evento. Gli eventi vengono accodati seguendo un ordine temporale: da quello più imminente a quello più lontano. Il gestore di un evento è tipicamente il gestore di un interruzione (nell'astrazione hardware dei componenti), il quale segnala *eventi* e richiama *comandi TinyOS*. Dopo ogni *evento Tossim*, il simulatore esegue i task della coda fino a che questa non è vuota. Questo tipo di soluzione non aderisce al comportamento del TinyOS. In effetti viene eliminata in questo modo il diritto di prelazione che hanno i gestori degli eventi TinyOS sui *task*.

4.2.2 Modello radio

Tossim fornisce un semplice ma efficace modello per simulare la connettività dei nodi. Questo consiste in un grafo, dove ciascun nodo rappresenta un nodo sensore e gli archi rappresentano le connessioni tra i nodi. Gli archi sono unidirezionali e a ciascuno di essi è associato un numero che rappresenta la *Bit Error Rate* (BER). In pratica un arco da un nodo x a un nodo y (x,y), rappresenta la probabilità di alterazione di un bit a cui è soggetta la trasmissione di un pacchetto dal nodo x verso y, e non il contrario. Tossim

permette la costruzione di questo grafo o utilizzando strumenti esterni o specificandone le caratteristiche in un file secondo una ben definita sintassi. È previsto inoltre un modello dove ogni nodo può comunicare con tutti gli altri senza che si verifichino perdite di pacchetti a causa del mezzo trasmissivo.

Anche se questo modello non è in grado di simulare la propagazione delle onde radio, riproduce alcuni problemi che si verificano nella trasmissione dei pacchetti, come il mancato riconoscimento del simbolo di start, la corruzione dei dati, etc.

4.2.3 Livello Data-Link

Tossim simula il comportamento del sistema di comunicazione a livello di bit. In particolare è in grado di riprodurre il problema del nodo nascosto e può simulare errori in tutte le fasi della ricezione del pacchetto. Il problema del nodo nascosto si verifica quando due nodi , a e b, intendono comunicare con un terzo nodo c, ma entrambi non sono in grado di ascoltare le trasmissioni dell'altro. Se a e b inviano contemporaneamente un pacchetto a c, quello che ne risulta è un segnale disturbato. Tuttavia il modello simulato non prende in considerazione tutti i possibili problemi che possono sopraggiungere durante una trasmissione e che riguardano il riconoscimento del segnale, la sincronizzazione, la codifica dei dati, il mancato riscontro, etc.

4.2.4 Consumo energetico

Tossim non fornisce nessuna informazione sul consumo energetico da parte dei nodi tuttavia esistono degli strumenti che ne permettono il calcolo come PowerTossim. Diversamente viene fornita una tecnica, applicabile utilizzano Tossim per il calcolo dei consumi. Questa tecnica consiglia di annotare lo stato dei componenti che consumano energia durante tutta la simulazione e poi applicare il modello energetico calcolando il consumo totali a fine simulazione. Tuttavia qualsiasi tecnica adottata risulta leggermente imprecisa perché Tossim non modella il tempo di esecuzione del processore

4.3 Architettura

L'architettura del Tossim è composta da cinque parti:

- il supporto per la compilazione delle applicazioni TinyOS per il simulatore. Il compilatore `ncc` è in grado di compilare applicazioni sia per l'hardware che per il simulatore, modificando alcune opzioni;
- la cosa degli eventi discreti;
- una serie di reimplementazioni di componenti relative all'hardware di basso livello. Questo permette la sostituzione dei componenti di basso livello in base alla piattaforma per cui vengono compilate (pc, mica, micadot);
- un meccanismo che permette di modellare la connettività tra i nodi e il fenomeno da analizzare;
- i servizi di comunicazione che permettono ad applicazioni esterne di interagire con la simulazione.

4.4 Servizi di comunicazione

Tossim fornisce dei meccanismi che consentono ad un programma esterno di visualizzare lo stato della simulazione, di controllarne alcuni parametri e di comunicare con i nodi utilizzando una connessione TCP/IP. La comunicazione tra il simulatore e l'applicazione esterna è definita da un protocollo che regola lo scambio di eventi e comandi. Tipici eventi sono i messaggi di debug che il programmatore inserisce nel codice, l'invio di pacchetti radio, le letture dei sensori, etc. I comandi invece possono spegnere o accendere un determinato nodo, iniettare pacchetti radio nella rete, e altri.

4.4.1 Tinyviz

Tinyviz è uno strumento per controllare e visualizzare lo stato della simulazione. È interamente realizzato in Java e fornisce un interfaccia grafica che permette facilmente di modificare i parametri della simulazione. In più è in grado di visualizzare la rete di sensori al completo e di evidenziare alcune azioni dei nodi (ad esempio lo spegnimento e l'accensione di un led o la trasmissione di un pacchetto). Di per se Tinyviz offre poche possibilità di controllare la simulazione. Si può ad esempio rallentare la simulazione, fermarla e farla ripartire, oppure spegnere o riaccendere un nodo. La potenza di questo strumento risiede nei *plug-in* che forniscono funzionalità più complesse all'utente. Per esempio esistono *plug-in* per impostare la posizione dei nodi, scegliere un particolare modello per le connessioni radio, etc. nel pacchetto di installazione del Tinyviz vengono forniti di una serie di *plug-in* che possono essere attivati o disattivati durante la simulazione. Tuttavia viene messa a disposizione dello sviluppatore un interfaccia per realizzare nuovi *plug-in* e quindi nuove funzionalità

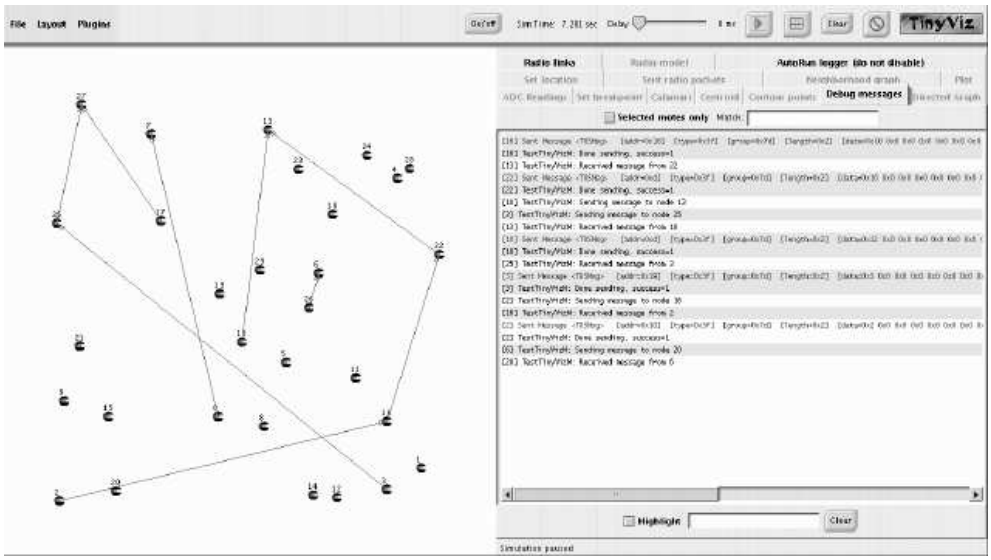

Figura 4.1: Tinyviz.

Capitolo 5

Architettura del simulatore *ma*-Tossim

Il nostro progetto consiste nel realizzare un simulatore, basato su Tossim, che sia in grado di simulare il comportamento di più applicazioni che cooperano per il raggiungimento di determinati obbiettivi. Questo capitolo intende fornire una descrizione dell'architettura del simulatore da noi proposto, il *ma*-Tossim , partendo dalla descrizione del protocollo, che fa parte dell'implementazione di Tossim e Tinyviz, che ci ha fornito lo spunto per il nostro lavoro.

5.1 Protocollo di comunicazione

La comunicazione tra Tossim e un'applicazione esterna è definita da un protocollo noto come *Tossim-Tinyviz Protocol*. Come il nome suggerisce questo protocollo nasce dall'esigenza di permettere ad un interfaccia grafica di visualizzare e soprattutto controllare l'ambiente di simulazione mediante un ben definito insieme di messaggi. Per un applicazione Java che intende utilizzare questo protocollo esiste una classe che lo implementa, disponibile nella cartella tinyos-1.x/tools/java/net/tinyos/sim e denominata `SimProtocol.java`

5.1.1 I canali di comunicazione

Tossim al suo avvio, e prima di inizializzare i parametri della simulazione, lancia due thread che si pongono in ascolto ciascuno su una determinata porta: la *command port* (10584) e la *event port* (10585). Un numero finito di *client* possono collegarsi a tali porte e il valore massimo è specificato dalla macro `#define MAX_CLIENT_CONNECTIONS`, definita nel file /tinyos-1.x/tos/platform/pc/external_comm.h e che attualmente è impostata a quattro. Eventualmente, modificando tale valore, si possono aumentare il numero di *client* in grado di connettersi al simulatore.

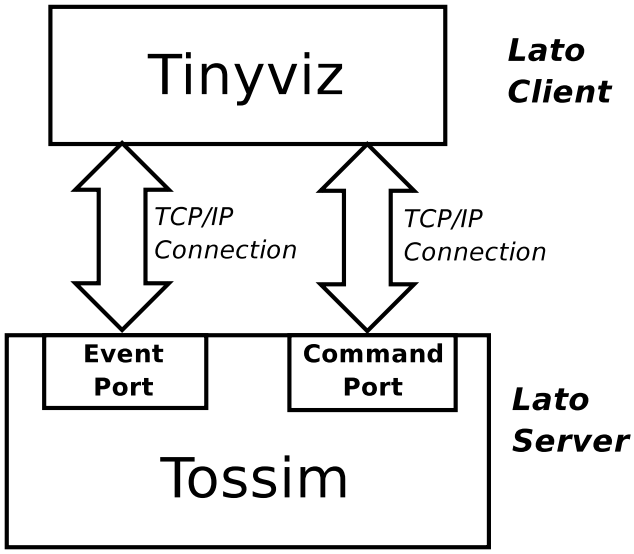

Figura 5.1: Modello di comunicazione.

Le connessioni sulla *event port* consentono ai *client* di leggere gli eventi prodotti dal simulatore. Per ogni evento prodotto il simulatore attende la ricezione di un riscontro che consiste nell'invio di un singolo byte da parte del *client*. In questo modo è possibile rallentare la velocità della simulazione, ritardando l'invio del riscontro. Nel caso vi fossero più connessioni aperte con la *event port* il simulatore attenderà un riscontro per ogni client prima di proseguire. In conclusione le applicazioni che utilizzano questa porta sono in grado di visualizzare lo stato della simulazione e di controllarne la velocità.

Le connessioni sulla *command port* consentono l'invio di messaggi che modificano alcuni parametri della simulazione e a differenza degli eventi non necessitano di riscontri.

5.1.2 I messaggi

Il formato dei messaggi che il simulatore è in grado di generare (gli eventi) o comprendere (i comandi) è descritto nel file /tinyos-1.x/tos/platform/pc/GuiMsg.h. Il codice seguente descrive l'intestazione dei messaggi:

```
typedef struct GuiMsg {
 uint16_t msgType;
  uint16_t moteID;
  long long time;
  uint16_t payLoadLen;
} GuiMsg;
```

Il campo

- `msgType`
  indica che tipo di informazione trasporta il messaggio;
- `moteID`
  contiene l'identificativo del nodo che ha prodotto il messaggio nel caso di un evento o del nodo a cui è destinato, nel caso di un comando;
- `time`
  rappresenta l'istante di tempo nel quale è stato prodotto il messaggio nel caso di un evento;
- `payLoadLen`
  informazione sulla lunghezza del campo dati.

All'intestazione può seguire o meno il campo dati a seconda del tipo di messaggio.

Eventi

Gli eventi prodotti dal Tossim per un applicazione esterna non hanno nulla in comune con gli *eventi Tossim*, né con gli *eventi TinyOS* citati in precedenza, e da qui in poi verranno chiamati *eventi Tinyviz* per una migliore distinzione. Nel file /tinyos-1.x/tos/platform/pc/GuiMsg.h viene definita una variabile per distinguere i possibili eventi, di cui ne riportiamo un frammento.

```
enum {
  /* Events */
  AM_DEBUGMSGEVENT,
  AM_RADIOMSGSENTEVENT,
  AM_UARTMSGSENTEVENT,
  AM_ADCDATAREADYEVENT,
  AM_TOSSIMINITEVENT,

  .....
};
```

Diamo ora una breve descrizione di ciascuno di essi:

- `AM_DEBUGMSGEVENT`
  Questo tipo di evento contiene nel campo dati una stringa prodotta dalla macro `dbg()`. Ad esempio il seguente codice,

  ```
  dbg(DBG_BOOT, "AM Module inizialized\n")
  ```

  inserito nel codice del componente /tinyos-1.x/tos/system/AMStandard.nc, produce, all'inizializzazione del modulo AM (il modulo radio), un messaggio tipo `AM_DEBUGMSGEVENT`, con la stringa

  ```
  "AM Module inizialized\n"
  ```

  nel campo dati. In pratica tutti i messaggi di *debug* inseriti nel codice, se abilitati, producono un *evento Tinyviz*.

- `AM_RADIOMSGSENTEVENT`
  Questo tipo di evento viene generato da un nodo che ha trasmesso un pacchetto via radio. Nel campo dati è memorizzato il contenuto dell'intero pacchetto, sia l'*Header* che il campo *Data*, il quale non è affetto da errori, cioè anche se si utilizza un modello radio con possibili alterazioni dei pacchetti, queste non vengono propagate sugli *eventi Tinyviz*.

- `AM_UARTMSGSENTEVENT`
  Quando un nodo invia un pacchetto sulla sua porta seriale, il simulatore produce in contemporanea un *evento Tinyviz* che contiene, al suo interno, il pacchetto stesso.

- `AM_ADCDATAREADYEVENT`
  In seguito alla lettura di un dato su uno dei canali del convertitore analogico/digitale (ADC) viene prodotto un evento che contiene il valore letto.

- `AM_TOSSIMINITEVENT`
  Questo tipo di evento viene generato in fase di inizializzazione dell'ambiente di simulazione e informa l'applicazione sul numero di nodi di cui è composta la rete.

I Comandi

I comandi, al contrario degli eventi, sono prodotti dalle applicazioni per controllare l'ambiente di simulazione. Anche in questo caso i *comandi TinyOS* non hanno nessun tipo di relazione con questo tipo comandi che indicheremo, dove necessario per non confondere le idee, con il nome di *comandi Tinyviz*. La stessa variabile che permette di distinguere gli eventi è utilizzata per i *comandi Tinyviz* e di cui ora ne daremo la forma completa.

```
enum {
  /* Events */
```

```
  AM_DEBUGMSGEVENT,
  AM_RADIOMSGSENTEVENT,
  AM_UARTMSGSENTEVENT,
  AM_ADCDATAREADYEVENT,
  AM_TOSSIMINITEVENT,

  /* Commands */
  AM_TURNONMOTECOMMAND,
  AM_TURNOFFMOTECOMMAND,
  AM_RADIOMSGSENDCOMMAND,
  AM_UARTMSGSENDCOMMAND,
  AM_SETLINKPROBCOMMAND,
  AM_SETADCPORTVALUECOMMAND,
};
```

Daremo ora una breve descrizione sul significato di ogni comando.

- `AM_TURNONMOTECOMMAND`
  Questo tipo di comando permette di riaccendere in nodo che era stato precedentemente spento.

- `AM_TURNOFFMOTECOMMAND`
  Con questo comando siamo in grado di spegnere un nodo della rete.

- `AM_RADIOMSGSENDCOMMAND`
  Un applicazione esterna può iniettare pacchetti radio nella rete di sensori grazie a questo tipo di messaggio. Il pacchetto arriverà ad un qualsiasi nodo senza subire alterazioni.

- `AM_UARTMSGSENDCOMMAND`
  In modo simile al caso precedente, è possibile inviare un pacchetto sulla porta seriale di un nodo utilizzando questo comando.

- `AM_SETLINKPROBCOMMAND`
  Inviando tale messaggio è possibile impostare il valore della probabilità

che un bit possa essere alterato durante la trasmissione di un pacchetto da un nodo *a* ad un nodo *b*.

- `AM_SETADCPORTVALUECOMMAND`
  Questo tipo di messaggio permette di impostare il valore letto dal convertitore analogico/digitale su un determinato canale.

Questi sono gli unici messaggi (sia eventi che comandi) che il simulatore è in grado di manipolare. Il loro numero non è elevato perché questo consente di semplificare al massimo il protocollo e quindi la gestione delle comunicazioni. Infatti gli sviluppatori diffidano dall'aggiungere nuovi tipi di messaggi, anche perché si dovrebbe modificare l'implementazione del protocollo in svariati punti. Nel caso avessimo bisogno di aggiungerne degli altri, suggeriscono, se possibile, di incapsularli in quelli già esistenti, prima di modificare il protocollo di simulazione.

5.2 Architettura software

Il motivo che ci ha spinti nella scelta di implementare un applicazione esterna per poter simulare contemporaneamente più applicazioni, è dovuto alla possibilità di controllare parte della simulazione mediante un protocollo ben definito. Grazie agli *eventi Tinyviz* infatti, siamo in grado di conoscere lo stato del sistema con una certa precisione, dato che, la generazione di tali eventi copre i punti chiave della simulazione con un buon livello di dettaglio. La precisione assoluta si avrebbe solo nel caso in cui si produrrebbe un *evento Tinyviz* per ogni *evento Tossim*. Inoltre, per mezzo dei comandi, possiamo modificare alcuni parametri della simulazione ed interagire con la stessa.

5.2.1 I componenti

L'architettura software del nostro progetto consiste dei seguenti componenti:

- un certo numero di istanze di Tossim, dove ciascuna rappresenta la simulazione di un applicazione TinyOS,

- un istanza del Tinyviz, che permette di visualizzare e controllare l'ambiente di simulazione nella sua interezza,
- un modulo, che chiameremo Pivot e rappresenta la nostra implementazione. Questo modulo sta nel mezzo tra le varie istanze di Tossim e l'applicazione Tinyviz, e ha il compito di rendere consistente la simulazione e permettere all'utente di controllare e visionare lo stato della simulazione.

I componenti della architettura comunicano tra loro per mezzo di connessioni TCP/IP La figura 5.2 descrive quanto detto sopra.

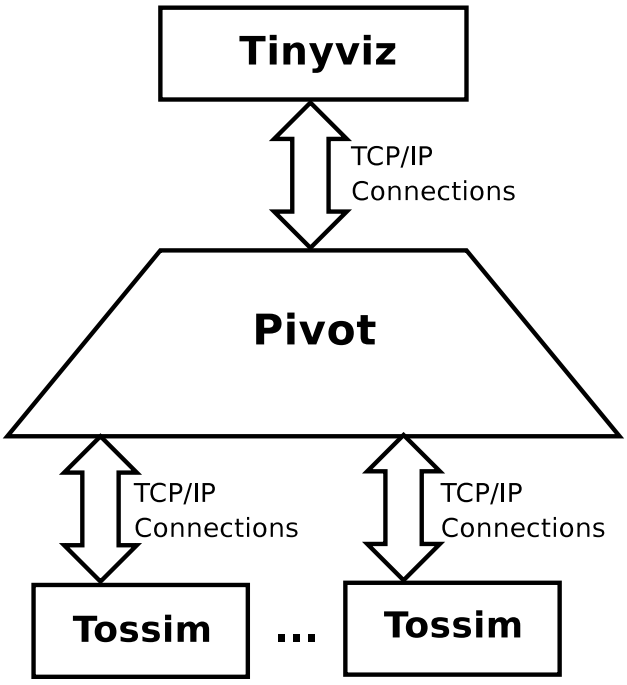

Figura 5.2: Componenti dell'architettura

5.2.2 Il modulo Pivot

Poiché cercheremo il più possibile di mantenere inalterate le implementazioni di Tossim e di Tinyviz ci concentriamo ora sugli obbiettivi che il nostro

modulo dovrà realizzare. In particolare il modulo Pivot:

- *sincronizzare le varie istanze di Tossim*
  Poiché ciascuna applicazione da simulare è un processo, che non è stato progettato per collaborare con le altre simulazioni, risulta necessario realizzare un meccanismo, che sfruttando le possibilità offerte, riesca a sincronizzare le applicazioni.

- *essere "trasparente" rispetto agli eventi Tinyviz*
  Il modulo dovrà coinvogliare gli eventi provenienti dalle varie applicazioni verso l'esterno, cioè verso Tinyviz, seguendo l'ordinamento temporale.

- *essere "trasparente" rispetto ai comandi Tinyviz*
  Esistono altre applicazioni, oltre al Tinyviz, che sono in grado di generare comandi da sottoporre al simulatore. Il nostro modulo dovrà permettere ad una qualsiasi di queste applicazioni di inviare comandi e garantire che questi arrivino al giusto destinatario.

- *mettere in comunicazione i nodi appartenenti ad applicazioni diverse*
  Affinché la simulazione si svolga in modo corretto, si dovranno attuare dei meccanismi per mettere in comunicazione i nodi appartenenti ad applicazioni diverse.

- *mantenere inalterato il protocollo Tossim-Tinyviz*
  Le soluzioni che implementeremo dovranno essere aderenti al protocollo in questione sfruttandone al massimo tutte le potenzialità, senza apportare modifiche di alcun genere.

Capitolo 6

Progetto in dettaglio

In questo capitolo analizzeremo le caratteristiche della soluzione da noi proposta sulla base degli obbiettivi discussi in precedenza per la realizzazione del modulo Pivot (vedi § 5.2.2)e ne descriveremo l'architettura. Inoltre metteremo in luce alcuni aspetti dell'implementazione che riteniamo importanti per comprendere il funzionamento del del *ma*-Tossim.

6.1 Modifiche al codice del simulatore

Nonostante i nostri propositi, si è reso necessario modificare del codice per il seguente motivo. Il simulatore Tossim, come abbiamo visto nel § 5.1, lancia due *thread* che si pongono in ascolto su due porte, la *command port* (10584) e la *event port* (10585), in attesa che qualche applicazione esterna si connetta a tali porte. Se volessimo lanciare più istanze del simulatore Tossim, la prima di queste si avvierebbe senza alcun problema, le successive però, in fase di inizializzazione, troverebbero le porte utilizzate da protocollo Tossim-Tinyviz già occupate. Questo ci impedisce di lanciare in simulazione più applicazioni alla volta. Risulta quindi inevitabile dover apportare delle modifiche al codice per poter lanciare più simulazioni che non entrino in conflitto sulle stesse porte.

In seguito ad una ricerca, siamo riusciti a scoprire che il file /tinyos-1.x/tos/platform/pc/external_comm.h contiene le dichiarazioni delle porte da usare:

```
...
#include <pthread.h>

#define COMMAND_PORT 10584
#define EVENT_PORT 10585

#define MAX_CLIENT_CONNECTIONS 4
....
```

Il nostro scopo è quello di modificare tale file in modo da associare ad istanze di Tossim, che dovranno essere eseguite contemporaneamente, porte diverse. In particolare, sapendo quali applicazioni dovranno essere simulate, potremmo associare ad ognuna di queste, nella fase di compilazione, delle porte predefinite e distinte.

La soluzione adottata prevede la seguente modifica al codice del file /tinyos-1.x/tos/platform/pc/external_comm.h:

```
...
#include <pthread.h>

// inizio modifica

#ifndef Multi
#define COMMAND_PORT 10884
#define EVENT_PORT 10885
#endif


#ifdef App1
```

```
#define COMMAND_PORT 11884
#define EVENT_PORT 11885
#endif


#ifdef App2
#define COMMAND_PORT 12884
#define EVENT_PORT 12885
#endif

#ifdef App3
#define COMMAND_PORT 13884
#define EVENT_PORT 13885
#endif

#ifdef App4
#define COMMAND_PORT 14884
#define EVENT_PORT 14885
#endif

#ifdef App5
#define COMMAND_PORT 15884
#define EVENT_PORT 15885
#endif

// fine modifica

#define MAX_CLIENT_CONNECTIONS 4
...
```

Come possiamo notare, le modifiche apportate ci permettono di simulare contemporaneamente fino a cinque applicazioni TinyOS, specificando degli

opportuni argomenti da passare al compilatore. Comunque il limite delle cinque applicazioni è facilmente superabile aggiungendo del codice in base allo schema riportato sopra.

Notare come, nel caso volessimo simulare più applicazioni alla volta, passando al compilatore l'argomento App1, viene creata un'applicazione a cui vengono associate le porte 11884 (*command port*) e 11885 (*event port*); nel caso si passasse l'argomento App1, viene creata un applicazione in ascolto sulle porte 12884 (*command port*) e 12885 (*event port*) e cosí via. Se invece non viene passata nessuna delle cinque opzioni (App1, App2, App3, App4, App5) viene generata un applicazione che utilizza le porte del protocollo Tossim-Tinyviz, cioè 10884 per la *command port* e 10885 per la *event port*.

Contestualmente si è reso necessario modificare anche il `Makerules` per adattarlo alla nuova configurazione, il quale viene inserito nel file `Makefile` che specifica quali applicazioni TinyOS saranno simulate contemporaneamente.

Se volessimo quindi, simulare applicazioni diverse in un unico ambiente, dovremo realizzare un Makefile secondo i seguente schema:

```
COMPONENT1=<nome della prima applicazione>
PFLAGS1=<opzioni aggiuntive da passare al compilatore ncc
        relative alla prima applicazione>
APP_DIR1=<percorso assoluto della cartella dei file della
          prima applicazione>

COMPONENT2=<nome della seconda applicazione>
PFLAGS2=<opzioni aggiuntive da passare al compilatore ncc
        relative alla seconda applicazione>
APP_DIR2=<percorso assoluto della cartella dei file della
          seconda applicazione>

...
```

```
COMPONENT5=<nome della quinta applicazione>
PFLAGS5=<opzioni aggiuntive da passare al compilatore ncc
        relative alla quinta applicazione>
APP_DIR5=<percorso assoluto della cartella dei file della
          quinta applicazione>

include ../Makerules
```

avendo cura di inserire il file Makerules nella cartella superiore. In questo modo lanciando il comando make ci troveremo nella stessa cartella gli eseguibili delle applicazioni da simulare.

6.2 Suddivisione del modulo in *thread*

Il nostro modulo è stato suddiviso in vari *thread* per gestire la meglio la concorrenza insita in questo simulatore. Di seguito riassumeremo le caratteristiche di ciascun *thread*.

6.2.1 *Thread* Pivot

Questo *thread* è il componente principale del simulatore *ma*-Tossim ed assolve tutte le funzioni principali per garantire una corretta simulazione. Questo *thread* ha il compito di:

- sincronizzare le istanze di Tossim;
- inviare gli eventi all'applicazione Tinyviz;
- inviare i comandi alle applicazioni simulate;
- inviare i pacchetti generati da un nodo e destinati a nodi appartenenti ad applicazioni diverse.

Comunque un descrizione dettagliata del funzionamento di questo componente verrà sviluppata nei prossimi paragrafi

6.2.2 Il servente delle applicazioni esterne

Questo *thread* si pone in ascolto sulla porta dei comandi in attesa che un applicazione esterna si connetti a tale porta. L'obbiettivo principale è quello di servire tali applicazioni e smistare le richieste verso l'effettivo gestore dei comandi. Questo *thread* è in grado di accettare fin ad un massimo di quattro connessioni.

6.2.3 Il *thread* lettore dei comandi

Questo *thread* si occupa della:

1. lettura dei comandi provenienti da un'applicazione;
2. riconoscimento i comandi che modificano il modello radio;
3. inserimento dei comandi in una lista;

Quando viene riconosciuto un comando tipo `AM_SETLINKPROBCOMMAND` bisogna analizzarlo per modificare il modello radio interno al modulo Pivot. Questo *thread* quindi, una volta riconosciuto il comando lo passa a delle specifiche funzioni che permettono l'aggiornamento del modello radio.

Per quanto riguarda tutti i comandi, questi vengono inseriti in una lista che verrà esaminata tra la gestione di un evento ed un altro.

Questo tipo *thread* viene creato dal servente delle applicazioni esterne ogni volta che si stabilisce una connessione con la *command port*.

6.2.4 Il *thread* lettore degli eventi

Il lettore degli eventi esegue ciclicamente i seguenti passi:

1. legge gli eventi provenienti da un applicazione;
2. inserisce ogni evento letto in un *buffer* condiviso con il *thread* Pivot;
3. si blocca un attesa che l'evento letto venga inviato all'applicazione Tinyviz;

4. una volta sbloccato invia il riscontro all'istanza del Tossim dal quale ha ricevuto l'evento

In fase di inizializzazione viene creato un *thread* di questo tipo per ogni applicazione che viene simulata, il quale poi si connette alla *event port* dalla quale riceve gli *eventi Tinyviz*

6.3 Mappatura dei nodi

Per mappatura dei nodi intendiamo come l'identificativo dei nodi simulati da ciascuna applicazione viene tradotto nell'identificativo di nodi dell'intera simulazione. Chiariamo meglio. Ogni applicazione simula un certo numero di nodi e ad ogni nodo viene assegnato un valore che lo identifica all'interno della rete di sensori. Questo identificativo è un numero intero che viene assegnato a partire dal valore zero. Ad esempio, se volessimo simulare un applicazione con cinque nodi, Tossim assegnerà ad ogni nodo un identificativo che corrisponderà ad un valore compreso tra zero e quattro. Saranno simulati così il nodo 0, il nodo 1, il nodo 2, il nodo 3 e il nodo 4. Questo identificativo ha una duplice funzione: oltre a nominare i nodi della simulazione rappresenta anche l'indirizzo di rete del nodo. Se il nodo quattro, ad esempio, volesse spedire un pacchetto al nodo 3 dovrebbe specificare come indirizzo del destinatario il valore tre.

Nel nostro caso abbiamo bisogno di simulare più applicazioni contemporaneamente, quindi ci troveremo ad affrontare la questione dei nodi con indirizzo simile, cioè esisterebbero dei nodi appartenenti ad applicazioni diverse che però condividono lo stesso identificativo. Le soluzioni che è possibile adottare sono due. La prima traduce all'interno del modulo Pivot l'indirizzo di ogni nodo in un indirizzo fittizio. La seconda soluzione non prevede nessuna traduzione, ma gli identificativi associati a ciascun nodo del Tossim rimangono inalterati quando vengono trattati all'interno del *ma*-Tossim.

Da semplici considerazioni si capisce che la prima soluzione è infattibile. Infatti se effettuiamo una traduzione degli indirizzi allora dovremo modificare

il campo `moteID` dei messaggi, sia eventi che comandi. Tuttavia nel caso dei messaggi radio si potrebbe verificare il caso in cui all'interno del campo dati ci sia un riferimento ad un nodo (ad esempio nel caso dell'instradamento dei pacchetti i nodi potrebbero scambiarsi informazioni circa la tabella dei nodi vicini). In questo caso il nostro modulo dovrebbe esaminare il campo dati di ogni pacchetto ed effettuare le traduzioni di indirizzo corrette. Questa strada è ovviamente impraticabile poiché non conosciamo a priori che tipo di informazioni sono contenute in un pacchetto e quindi non potremo conoscere quali sono le modifiche che effettivamente bisognerebbe apportare. Quindi non ci rimane che l'altra strada.

La seconda soluzione prevede che non vengano modificati gli identificativi dei nodi simulati da ciascuna applicazione. Ad esempio, poniamo il caso di voler simulare due applicazioni, una con due nodi e l'altra con tre nodi, quindi in totale avremo cinque nodi da simulare. Applicando la convenzione utilizzata da Tossim, all'interno del nostro simulatore dovremo lanciare cinque nodi con indirizzo rispettivamente 0, 1, 2, 3 e 4, assegnando, per esempio, gli indirizzi 0 e 1 ai nodi simulati dalla prima applicazione e gli indirizzi 2, 3, e 4, ai nodi della seconda applicazione. Questo tipo di scelta ci permette di lanciare la prima applicazione con due nodi da simulare e la seconda applicazione con cinque nodi da simulare. C'è però il problema del conflitto di indirizzi per i nodi 0 e 1 delle due applicazioni, in più noi vorremo simulare all'interno della seconda applicazione non cinque nodi ma tre. Comunque se riuscissimo a spegnere i nodi 0 e 1 della seconda applicazione ogni problema sarebbe risolto. Ma ciò è possibile perché il simulatore mette a disposizione il comando `AM_TURNOFFMOTECOMMAND` che permette di spegnere qualsiasi nodo. In questo caso non si presenta più il problema di dover modificare gli indirizzi dei nodi, né all'interno del campo `moteID` ne tantomeno quelli all'interno del campo dati.

6.4 Il modello radio

Il modulo Pivot provvede a creare autonomamente un modello radio consistente con il modello del Tossim. Come per il Tossim, il modello radio generato dal simulatore *ma*-Tossim si compone di un grafo i cui nodi rappresentano i nodi simulati, mentre gli archi rappresentano i collegamenti radio tra i nodi sensore. Ad ogni arco viene assegnato un valore compreso tra zero e uno che rappresenta la probabilità che un bit venga alterato nella trasmissione del pacchetto. I collegamenti sono unidirezionali come nel caso del Tossim. cioè un arco che va da a a b a cui è associato il valore x indica che la probabilità che un bit venga alterato quando a trasmette un pacchetto a b vale x. L differenza tra i due grafi sta proprio negli archi che compongono il grafo. Non esistono cioè archi che vanno da un nodo a ad un nodo b se entrambi appartengono alla stessa applicazione. Il nostro simulatore deve infatti gestire le comunicazioni appartenenti ad applicazioni diverse mentre le trasmissioni tra nodi appartenenti alla stessa applicazione vengono gestiti direttamente dal Tossim.

6.5 Sincronizzazione delle applicazioni simulate

Il primo problema che si è dovuto affrontare nell'implementazione del *ma*-Tossim è la sincronizzazione temporale delle applicazioni TinyOS. Il simulatore infatti non è stato realizzato per condividere le proprie risorse con altri processi e quindi non siamo in grado di accedere alla coda degli *eventi Tossim* per recuperare le informazioni temporali sull'andamento della simulazione Le uniche informazioni sull andamento temporale sono contenute nel campo `time` degli *eventi Tinyviz*. Come gia descritto nel § 5.1.2 ogni evento porta con se l'informazione sull'istante temporale in cui è stato generato. In questo modo possiamo realizzare una coda, che chiameremo coda degli *eventi Tinyviz*, in cui verranno inseriti gli eventi provenienti da tutte le applicazioni.

Naturalmente gli elementi di tale coda verranno riordinati secondo un ordine temporale, dal più imminente a quello meno.

Di seguito riportiamo il codice della funzione che si occupa dell'inserimento in coda degli *eventi Tinyviz*:

```
synchronized public void addEvent(int i, TossimEvent e) {
  events[i] = e;
  if (e instanceof RadioMsgSentEvent)
     lre.insert(i, e.getTime());
  sem.signalAddEvent();
  while (events[i] != null) {
     try {
        wait();
     } catch (Exception ex) {
        System.err.println("EventBuffer: wait()
                            failed!" + ex);
     }
  }
}
```

Le righe:

```
synchronized public void addEvent(int i, TossimEvent e) {
  events[i] = e;
 ...
```

definiscono la funzione `void addEvent(int i, TossimEvent e)`, che permette l'inserimento degli *eventi Tossim* in un *buffer*, `events[]` del tipo `TossimEvent`, e l'inserimento dell'evento `e` in tale buffer.

Di seguito, le righe

```
 ...
  if (e instanceof RadioMsgSentEvent)
     lre.insert(i, e.getTime());
```

```
        sem.signalAddEvent();
      ...
```

memorizzano, se l'evento contiene un pacchetto radio, l'istante di invio del pacchetto e l'"indirizzo" dell'evento. Poi viene segnalata la presenza di un evento in coda sbloccando eventuali *thread* in attesa.

Infine, le righe:

```
    ...
     while (events[i] != null) {
        try {
           wait();
        } catch (Exception ex) {
           System.err.println("EventBuffer: wait()
                                failed!" + ex);
        }
     }
  }
```

mostrano che il *thread* si blocca in attesa di essere risvegliato se l'evento inserito non è stato ancora elaborato, cioè quando la condizione `events[i] != null` risulta vera. Notiamo infine, come la semantica della *signal*, riattivi tutti i *thread* sospesi, quindi impone che la condizione d'uscita sia ritestata ad ogni risveglio

Il blocco del *thread* causa il blocco dell'applicazione simulata che ha inviato l'evento. In effetti, essendo questo *thread* l'interfaccia tra il modulo Pivot e l'applicazione TinyOS, si occupa di gestire il protocollo Tossim-Tinyviz, che come ricordiamo ci permette di rallentare o di mettere in pausa la simulazione. Quindi se l'operazione di inserimento dell'evento nel *buffer* viene effettuata tra la sua ricezione e l'invio del riscontro siamo in grado di sincronizzare le applicazioni.

Di seguito riportiamo un frammento di codice del corpo del *thread*:

```
  public void run() {
```

```
...
  while (true) {
    try {
      event = eventSimProtocol.readEvent();
    } catch (Exception e) {
      ...
    }
    ...
    events.addEvent(id, event);
    try {
      eventSimProtocol.ackEventRead();
    } catch (Exception e) {
      ...
    }
  }
}
```

Lo sblocco del *thread* avviene in seguito all'estrazione dell'evento:

```
synchronized public TossimEvent extractEvent(int i) {
  TossimEvent e = events[i];
  events[i] = null;
  sem.signalExtractEvent();
  notifyAll();
  return e;
}
```

e comporta l'invio immediato del riscontro e quindi l'avanzamento di un "passo" della simulazione.

6.6 Trasparenza del modulo Pivot

Il modulo Pivot deve essere trasparente rispetto sia ai comandi che agli eventi. Questo significa da un lato che ogni evento dovrà essere inviato verso

Tinyviz rispettando l'ordine temporale, dall'altra che qualsiasi comando ricevuto dovrà essere recapitato al giusto destinatario. Recapitare al giusto destinatario un comando significa che, se ad esempio un applicazione comandasse lo spegnimento di un nodo, dobbiamo essere in grado di distinguere a quale istanza di Tossim appartiene il nodo in questione e quale tra i nodi simulati è il destinatario del messaggio.

6.6.1 L'evento `AM_TOSSIMINITEVENT`

Questo tipo di evento viene generato nella fase di inizializzazione della simulazione. Tinyviz come primo evento si aspetta un messaggio di questo tipo che lo informi sul numero di nodi che si stanno simulando. In base a questo valore poi, Tinyviz mostra sul pannello della simulazione un numero di nodi pari a quello indicato nel messaggio.

Nel nostro caso però il numero di eventi `AM_TOSSIMINITEVENT` che il modulo Pivot riceve è pari al numero delle applicazioni, quindi deve essere posto un filtro, che blocchi gli eventi `AM_TOSSIMINITEVENT` e ne invii uno solo con il valore corretto.

6.6.2 Trasparenza rispetto agli eventi

Come abbiamo appena spiegato, Pivot deve essere in grado di inviare gli eventi ad un applicazione esterna in modo corretto. Ciò significa:

- rispettare l'ordinamento temporale,
- inviare un solo evento `AM_TOSSIMINITEVENT` con il valore del numero di nodi simulati corretto,
- inviare tutti gli altri tipi di eventi mantenendo inalterato il loro contenuto.

Il primo punto risulta soddisfatto dal modo in cui le applicazioni simulate si sincronizzano. I terzo punto è viene banalmente rispettato. Il secondo

punto richiede una particolare attenzione in fase di ricezione degli eventi. A tal proposito riportiamo un frammento di codice del *thread* realizza le funzionalità più importanti del modulo Pivot:

```
public void run() {
  ...
  while (true) {

    sem.waitEvents();

    idx = events.olderEvent();
    ...
    ev = events.readEvent(idx);
    Message msg = (Message) ev;
    if ((int) msg.amType() ==
        EventUtil.AM_TOSSIMINITEVENT) {

      ev = EventUtil.createInitEvent(
             EventUtil.AM_TOSSIMINITEVENT,
             ev.getMoteID(),
             ev.getTime(),
             totalMotes);

      msg = (Message) ev;

    }
    /* Write to TinyViz */
    try {
      dos.writeShort((int) msg.amType());
      dos.writeShort(ev.getMoteID());
      dos.writeLong(ev.getTime());
      int payload_len = msg.dataLength();
```

```
        dos.writeShort(payload_len);
        if (payload_len > 0)
    dos.write(msg.dataGet(),
                    msg.baseOffset(),
      msg.dataLength());

        dos.flush();
      } catch (Exception e) {
        ...
      }

      ...

      if ((int) msg.amType() ==
          EventUtil.AM_TOSSIMINITEVENT)
        events.deleteEvents();
      else
        events.extractEvent(idx);
  }
```

Le prime righe:

```
  public void run() {
    ...
    while (true) {

      sem.waitEvents();
      ...
```

indicano che si tratta del corpo del *thread* e che questo compirà ciclicamente le stesse operazioni all'infinito. All'interno del ciclo, viene verificata la presenza di eventi nel *buffer*. Se non ci fossero eventi in coda il*thread* si bloccherebbe fino all'arrivo di un nuovo evento.

Nelle righe successive:

```
...
idx = events.olderEvent();
...
ev = events.readEvent(idx);
...
```

viene prelevato l'evento più imminente e salvato nella variabile `ev`.

In seguito le linee

```
Message msg = (Message) ev;
if ((int) msg.amType() ==
    EventUtil.AM_TOSSIMINITEVENT) {

  ev = EventUtil.createInitEvent(
         EventUtil.AM_TOSSIMINITEVENT,
         ev.getMoteID(),
         ev.getTime(),
         totalMotes);

  msg = (Message) ev;

}
```

viene effettuata una conversione di tipi (dal tipo `TossimEvent` al tipo `Message`) che ci permette di verificare se il tipo di messaggio ricevuto è del tipo `AM_TOSSIMINITEVENT`. Se ciò dovesse risultare vero verrebbe creato un nuovo evento `AM_TOSSIMINITEVENT` con il valore corretto del numero di nodi da simulare (`totalMotes`) mentre il resto del messaggio rimane invariato (tipo messaggio, identificativo del nodo, istante di generazione). Quindi, il nuovo evento, viene riconvertito nel tipo `Message`, che fornisce una serie di utili funzioni per l'invio del messaggio all'esterno.

In seguito le righe:

```
        ...

        /* Write to TinyViz */
        try {
          dos.writeShort((int) msg.amType());
          dos.writeShort(ev.getMoteID());
          dos.writeLong(ev.getTime());
          int payload_len = msg.dataLength();
          dos.writeShort(payload_len);
          if (payload_len > 0)
    dos.write(msg.dataGet(),
                    msg.baseOffset(),
      msg.dataLength());

          dos.flush();
        } catch (Exception e) {
          ...
        }

        ...
```

si occupano di inviare il messaggio verso Tinyviz.

Infine l'ultimo frammento:

```
        ...

        if ((int) msg.amType() ==
            EventUtil.AM_TOSSIMINITEVENT)
          events.deleteEvents();
        else
          events.extractEvent(idx);
    }
```

elimina tutti gli eventi `AM_TOSSIMINITEVENT` in coda, oppure cancella l'evento appena esaminato. In tutti casi viene sbloccato uno o più *thread*, i quali faranno avanzare la simulazione di uno scatto.

6.6.3 Trasparenza rispetto ai comandi

Il nostro modulo deve permettere ad applicazioni esterne di poter inviare nel modo corretto comandi all'ambiente di simulazione. Questo significa che, per ogni comando ricevuto si dovrà:

- individuare l'applicazione a cui appartiene il nodo,
- individuare il nodo destinatario all'interno dell'applicazione,
- modificare l'identificativo del nodo destinatario del campo `moteID` del comando, con il valore corretto.

In funzione della mappatura scelta per i nodi simulati non c'è nessuna modifica da effettuare nel campo `moteID`, c'è invece la necessità di individuare l'applicazione giusta a cui appartiene il nodo. Infatti, il secondo punto si risolve da solo poiché, sempre in funzione della mappatura adottata, l'identificativo del nodo destinatario contenuto nel comando individua direttamente il nodo all'interno dell'applicazione.

La gestione dei comandi è molto semplice. Ogni comando ricevuto viene memorizzato in una struttura tipo lista dai *thread* preposti alla ricezione dei comandi. In seguito all'invio dei un evento verso l'applicazione Tinyviz, il *thread* Pivot controlla se tale lista non sia vuota. Se non è vuota allora vengono estratti tutti i comandi e inviati ai rispettivi nodi. Di seguito riportiamo un frammento di codice del *thread* Pivot relativo alla gestione dei comandi:

```
...

// Sending command to Tossim
while (!cmds.isEmpty()) {
```

```
    PivotCommand pc = cmds.extractCmd();
    TossimCommand tc = pc.toTossimCommand();
    try {
      if (pc.getAppID() != baseMote.length)
        cmdSimProtocol[pc.getAppID()].writeCommand(tc);
      else {
        for (int i = 0; i < numApps; i++)
          cmdSimProtocol[i].writeCommand(tc);
      }
    } catch (Exception e) {
      System.err.println("Pivot: Unable to send
                          command: " + e);
    }
  }

  ...
```

L'aspetto che ci interessa maggiormente di questo codice è la scelta dell'applicazione a cui inviare il comando. Come possiamo notare dalle righe:

```
  ...

      if (pc.getAppID() != baseMote.length)
        cmdSimProtocol[pc.getAppID()].writeCommand(tc);
      else {
        for (int i = 0; i < numApps; i++)
          cmdSimProtocol[i].writeCommand(tc);
      }

  ...
```

esistono due casi:

1. il comando viene inviato ad un applicazione specifica,

2. il comando viene inviato a tutte le applicazioni.

Questa distinzione nasce dal fatto che, nel caso si trattassero di comandi relativi alla gestione del modello radio, cioè comandi del tipo `AM_SETLINKPROBCOMMAND`, conviene inviarli a tutte le applicazioni piuttosto che ad una specifica. In effetti, per poter identificare l'applicazione verso cui è destinato questo tipo di comandi, dovremo analizzare il contenuto del campo dati di ciascuno di essi e poi in base ai nodi dell'arco, decidere se inviare il comando e a quale istanza di Tossim. In pratica, se l'arco coinvolge i nodi a e b e questi appartengono ad una stessa applicazione K allora ha senso inviare il comando ad K altrimenti no. Purtroppo queste operazioni sono abbastanza complesse da eseguire e potrebbero non portare effettivi vantaggi alla velocità di esecuzione. C'è da considerare poi il fatto che le modifiche al modello radio apportate durante al simulazione sono piuttosto rare, generalmente si impostano una volta e all'inizio della simulazione. Quindi si è preferito adottare una soluzione semplice ma ugualmente efficace anche in previsione di sviluppi futuri del simulatore *ma*-Tossim nei quali potrebbe essere utile che ogni applicazione conosca il modello radio completo.

Negli altri casi l'individuazione dell'applicazione è semplice e viene fatta in funzione del campo `moteID` del messaggio.

6.7 Comunicazione tra nodi appartenenti ad applicazioni diverse

Affinché la simulazione sia consistente, deve essere necessario mettere in comunicazione nodi che vengono simulati in applicazione diverse. Poiché la comunicazione tra nodi avviene mediante lo scambio di pacchetti via radio gli eventi che ci interessano sono quelli del tipo `AM_RADIOMSGSENTEVENT`, che ci informano che è stato inviato un messaggio via radio. Come abbiamo già visto, all'acquisizione di un evento viene notificato la ricezione di un evento tipo `AM_RADIOMSGSENTEVENT` se questo dovesse sopraggiungere:

```
synchronized public void addEvent(int i, TossimEvent e) {
  events[i] = e;
  if (e instanceof RadioMsgSentEvent)
     lre.insert(i, e.getTime());
  ...
}
```

cioè viene inserito in una lista (la lista degli eventi radio ricevuti), l'informazione sull'evento radio e sull'istante in cui è stato inviato. Per ridurre al massimo il ritardo tra la trasmissione del pacchetto e la sua ricezione, l'evento radio viene gestito non appena viene ricevuto. Infatti il *thread* Pivot dopo la gestione dell'evento più imminente e dell'invio degli eventuali comandi passa a controllare se ci sono eventi radio in attesa di essere presi in considerazione. Le operazioni effettuate da questo *thread* le riportiamo in questo frammento di codice:

```
...

// Radio packet
while (!radioEvents.isEmpty()) {
  RadioEventInfo info = radioEvents.extract();
  RadioMsgSentEvent radioMsg = getRadioEvent(info);
  sendRadioCommand(radioMsg, info);
}

...
```

Il ciclo esamina tutti i pacchetti presenti nella lista e li invia al nodo destinatario tramite la funzione:
`sendRadioCommand(radioMsg, info)`
Questa funzione ha due possibilità: o l'indirizzo del destinatario specificato nel pacchetto si riferisce ad un particolare nodo o, oppure il pacchetto viene inviato in *broadcast*.

Se viene inviato ad un nodo specifico allora le operazioni da svolgere sono molto semplici:

1. verifica se il modello radio utilizzato è soggetto ad alterazioni dei pacchetti;

2. se ciò è vero, estrae un numero a caso compreso tra zero e uno, e lo confronta con la probabilità che almeno un bit del pacchetto possa essere alterato durante la trasmissione, altrimenti passa al punto 4;

3. se tale numero è inferiore alla probabilità che il pacchetto arrivi alterato al destinatario, allora il pacchetto viene buttato via, altrimenti continua;

4. invia il pacchetto integro al destinatario.

Nel caso invece che, il pacchetto sia inviato in *broadcast* allora è necessario inviare il messaggio a tutti i nodi della rete. I particolare si dovranno ripetere le operazioni dal punto uno al punto quattro per:

- tutte le applicazioni, tranne quella da cui è partito il pacchetto;
- tutti i nodi di ciascuna applicazione.

6.8 Considerazioni sul modello di comunicazione

L'ambiente di simulazione realizzato da Tossim fornisce due semplici modelli per rappresentare il mezzo trasmissivo in cui si trova a lavorare una rete di sensori.

Tenendo conto che Tossim è in grado di inviare i pacchetti radio a livello di bit, il primo modello permette di modellare la probabilità che un bit venga alterato in fase di trasmissione. In pratica per ogni bit inviato, Tossim estrae un numero casuale per verificare in base alla *bit error rate* se tale

bit verrà modificato o meno. Il nostro simulatore per le comunicazioni tra applicazioni diverse utilizza un modello che considera i pacchetti invece dei bit. Questo però non modifica il comportamento di Tossim, in quanto è sufficiente il calcolo della probabilità che il pacchetto possa subire alterazioni, la quale dipende dalla lunghezza del pacchetto stesso e della *bit error rate* che sono informazioni che noi conosciamo.

Il secondo modello fornito da Tossim, garantisce che se si verificassero trasmissioni simultanee di due nodi che intendono comunicare con uno stesso nodo, i due pacchetti collide rebbero. Ciò è vero soprattutto quando si verifica il problema del nodo nascosto. Quindi, nel caso di una collisione, poiché il *CRC* dell'*header* del messaggio non coincide con quello effettivo, il pacchetto viene buttato via. Come abbiamo già ricordato, il *ma*-Tossim non è in grado di lavorare a livello di bit poiché ogni pacchetto inviato dal modulo Pivot ad una applicazione Tossim arriva integro al nodo ed entra direttamente dal livello *packet* della stratificazione dei componenti di rete. Quindi, eventuali collisioni che si verificano all'interno del modello simulato da Tossim non si propagano sul modello del *ma*-Tossim. Questo problema si potrebbe ovviare solo modificando al codice del simulatore. Una soluzione potrebbe essere quella di costringere Tossim a generare un particolare evento quando un nodo avvia la trasmissione di un pacchetto. In pratica, conoscendo l'istante di inizio trasmissione e i nodi coinvolti (sia mittente sia destinatario/i), potremmo capire se al termine dell'invio si sono verificate delle collisioni o meno, controllando che durante tutto questo periodo non ci siano state trasmissioni da parte di nodi vicini al nodo destinatario. D'altra parte sarebbe necessario anche un comando che forza una collisione per i nodi che appartengono ad una stessa applicazione. Ad esempio, poniamo il caso in cui un nodo a, appartenente ad un applicazione H, effettua la trasmissione di un pacchetto verso un nodo b, appartenente ad un applicazione K. Se durante la trasmissione si verifica una collisione, in seguito alla trasmissione da parte di un nodo c appartenente anch'esso a K, bisognerebbe forzare in qualche modo la collisione sul nodo b che altrimenti non avverrebbe poiché provocata in ap-

plicazioni diverse. Tuttavia nel caso se non si dovesse presentare il problema del nodo nascosto, ad esempio perché non si utilizza il modello radio *lossy*, la probabilità di collisione è molta bassa poiché lo garantisce il protocollo CSMA/CA del TinyOS.

Per quanto riguarda la velocità di simulazione, Tossim è un simulatore molto veloce che è in grado di simulare molte migliaia di nodi. L'aggiunta di una applicazione esterna che comunica con questo, ne rallenta la velocità . In effetti utilizzando Tinyviz con Tossim le prestazioni diminuiscono leggermente ma in maniera visibile. Il nostro modulo peggiora questa situazione in quanto è in comunicazione con più istanze del simulatore e con Tinyviz. Ma l'aspetto più interessante è che, tra la gestione di un evento ed un altro tutte le istanze del simulatore sono bloccate in attesa della ricezione di un riscontro. Una soluzione migliore potrebbe essere quella di realizzare una politica di gestione della coda degli eventi provenienti dal Tossim più sofisticata, per aumentare il livello di parallelismo. Ad esempio, si potrebbe accelerare la simulazione quando non vi sono pacchetti radio o comandi da gestire anticipando l'invio di un riscontro oppure si potrebbe far procedere in qualche modo in parallelo il modulo Pivot e un istanza del simulatore.

Conclusioni

In questa tesi è stato proposta e sviluppata una estensione di Tossim, per fornire un ambiente di simulazione per reti di sensori multi-applicazione chiamato *ma*-Tossim. L'obbiettivo principale è stato quello di realizzare uno strumento portabile che fosse in grado, per quanto possibile, di realizzare un ambiente di simulazione consistente con il modello fornito da Tossim.

Per quando riguarda la portabilità del software realizzato, le uniche modifiche apportate al codice di Tossim non coinvolgono le parti essenziali del suo funzionamento ma definiscono alcune macro per la scelta delle porte, sia quella degli eventi che quella dei comandi, su cui si pongono in ascolto le diverse istanze del simulatore. Lo strumento sviluppato, se non è selezionata l'opzione "multiapplicazione" si comporta come Tossim. Inoltre avendo utilizzato il linguaggio Java per l'implementazione del modulo Pivot possiamo eseguire il nostro simulatore su una qualsiasi piattaforma che supporti TinyOS e Java.

Possibili estensioni del simulatore *ma*-Tossim includono un supporto per il calcolo dei consumi energetici (avendo a disposizione gli eventi generati dal Tossim si potrebbe calcolare facilmente il consumo di energia di ciascun nodo e, ad esempio, visualizzare tale informazione in un *plug-in* del Tinyviz) e un modulo per lo spegnimento di un nodo che ha esaurito tutta la sua energia durante la simulazione.

Bibliografia

[1] S. Park, A. Savvides, and M. Srivastava, "Sensorsim: a simulation framework for sensor networks", in Proceedings of MSWiM'00, 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems.

[2] S. Sundresh, W. Kim, and G. Agha, "SENS: A sensor, environment and network simulator", in Proceedings of 37th Annual Simulation Symposium, 2004.

[3] L. F. Perrone and D. Nicol, "A scalable simulator for TinyOS applications", in Proceedings of WSC'02, Winter Simulation Conference, 2002.

[4] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and scalable simulation of entire TinyOS applications", in Proceedings of SenSys'03, First ACM Conference on Embedded Networked Sensor Systems, 2003.

[5] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "Emstar: a software environment for developing and deploying wireless sensor networks", in Proceedings of the USENIX Technical Conference, 2004.

[6] http://telegraph.cs.berkeley.edu/tinydb/

[7] A. Mainwaring, J. Polastre, R. Szewczyk, D Culler, e J. Anderson, "Wireless sensor network for habitat monitoring" presented at the 1st ACM

Int. Workshop Wireless Sensor Networks and Applications, Atlanta, GA, 2002.

[8] Smart buildings admit their faults. Lab Notes [Online] Available: http://www.coe.berkeley.edu/labnotes/1101smartbuildings.html

[9] A. N. Knaian, A wireless sensor network for smart roadbeds and intelligent transportation systems, Ph.D. dissertation, Mass. Inst.Technol., Cambridge, 1999.

[10] F. Siegemung. Smart-Its on the Internet – Integrating Smart Objects into the Everyday Communication Infrastructure. Technical note, September 2002. http://www.inf.ethz.ch/vs/publ/papers/smartits-demo-note-siegemund.pdf.

[11] M. Ogawa et al., Fully automated biosignal acquisition in daily routine through 1 month, International Conference on IEEE-EMBS, Hong Kong, 1998, pp. 1947-1950.

[12] J. Elson, L. Girod, and D. Estrin, Fine-grained network time synchronization using reference broadcasts, presented at the 5thSymp. Operating Systems Design and Implementation (OSDI), Boston, MA, 2002.

[13] S. Klemmer, S. Waterson, and K. Whitehouse. (2000, Dec.) Toward a location-based context-aware sensor infrastructure. [Online] Available: http://guir.berkeley.edu/projects/location/Location.pdf

[14] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar, Next century challenges: scalable coordination in sensor networks, in Proc.5th Annu. ACM/IEEE Int. Conf. Mobile Computing and Networking,1999, pp. 263-270.

[15] E. M. Royer and C.-K. Toh, A review of current routing protocolsfor ad-hoc mobile wireless networks, IEEE Pers. Commun., vol. 6, pp. 46-55, Apr. 1999.

[16] F. Zhao, J. Shin, and J. Reich, Information-driven dynamic sensor collaboration, IEEE Signal Processing Mag., vol. 19, pp. 61-72, Mar. 2002.

[17] Sinopoli, B.; Sharp, C.; Schenato, L.; Schaffert, S.; Sastry, S.S., Distributed Control Applications within Sensor Networks, Proceedings of IEEE, August 2003.

[18] Vijay Raghunathan, Curt Schurgers, Sung Park, Mani B Srivastava, Energy-aware wireless microsensor networks, IEEE Signal Processing Magazine , March 2002.

[19] C. Schurgers, O. Aberthorne, and M. B. Srivastava. Modulation Scaling for Energy Aware Communication Systems. In Intl. Symp. on Low Power Electronics and Design (ISLPED '01), pages 96-99, August 2001.

[20] A. Y. Wang, S. H. Cho, C. G. Sodini, and A. P. Chandrakasan. Energy Efficient Modulation and MAC for Asymmetric RF Microsensor Systems. In Intl. Symp. on Low Power Electronics and Design (ISLPED '01), pages 96-99, August 2001.

[21] A. Woo and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking, pages 221-235, Rome, Italy, July 2001. ACM.

[22] V. Kanodia, C. Li, A Sabharwal, B. Sadeghi, and E. W. Knightly. Distributed Multi-Hop Scheduling and Medium Access with Delay and Throughput Constraints. In Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking, pages 200-209, Rome, Italy, July 2001. ACM.

[23] L. Bao and J. J. Garcia-Luna-Aceves. A New Approach to Channel Access Scheduling for Ad Hoc Networks. In Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking, pages 210-220,Rome, Italy, July 2001. ACM.

[24] The network simulator. http://www.isi.edu/nsnam/ns/.

[25] http://www.alertsystems.org.

[26] A. Cerpa, J. Elson, M. Hamilton, J. Zhao, Habitat monitoring: application driver for wireless communications technology, ACM SIGCOMM'2000, Costa Rica, April 2001.

[27] http://www.greatduckisland.net/

[28] N. Noury, T. Herve, V. Rialle, G. Virone, E. Mercier, G. Morey, A. Moro, T. Porcheron, Monitoring behavior in home using a smart fall sensor, IEEE-EMBS Special Topic Conference on Microtechnologies in Medicine and Biology, October 2000, pp. 607-610.

[29] E.M. Petriu, N.D. Georganas, D.C. Petriu, D. Makrakis,V.Z. Groza, Sensor-based information appliances, IEEE Instrumentation and Measurement Magazine (December 2000) 31-35.

[30] I.A. Essa, Ubiquitous sensing for smart and aware environments, IEEE Personal Communications (October 2000).

[31] G. Hoblos, M. Staroswiecki, A. Aitouche, Optimal design of fault tolerant sensor networks, IEEE International Conference on Control Applications, Anchorage, AK, September 2000.

[32] J. Rabaey, J. Ammer, J.L. da Silva Jr., D. Patel, Pico- Radio: ad-hoc wireless networking of ubiquitous low-energy sensor/monitor nodes, Proceedings of the IEEE Computer Society Annual Workshop on VLSI, Orlanda, Florida, April 2000.

[33] A. Y. Wang, S. H. Cho, C. G. Sodini, and A. P. MAC for Asymmetric RF Microsensor Systems. In Intl. Symp. on Low Power Electronics and Chandrakasan. Energy Efficient Modulation and Design, August 2001.

[34] C. Schurgers, O. Aberthorne, and M. B. Srivastava. Modulation Scaling for Energy Aware Com-munication Systems. In Intl. Symp. on Low Power Electronics and Design, August 2001.

[35] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In Proceedings of SIGCOMM, september 1990.